DTIC FILE COPY

(4)

RADC-TR-88-166
Final Technical Report
July 1988

AD-A204 352

RECONFIGURABLE C² DDP SYSTEM

Carnegie-Mellon University

E. Douglas Jensen, Hideyuki Tokuda, John P. Lehoczky, Lui Sha, Raymond K. Clark, C. Douglass Locke, J. Duane Northcutt, Samuel Shipman, Charles Hitchcock, H. M. Brinkley Sprunt, Ragunathan Rajkumar, James Wendorf, Roli Wendorf, and Andre M. Van Tilborg

DTIC
ELECTE
FEB 1 3 1989
S D
D

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

89 2 13 286

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-88-166 has been reviewed and is approved for publication.

APPROVED: Thomas F. Lawrence

THOMAS F. LAWRENCE
Project Engineer

APPROVED: Raymond P. Urtz Jr.

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: James W. Hyde III

JAMES W. HYDE, III.
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notice on a specific document requires that it be returned.

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) 5879	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-166
6a. NAME OF PERFORMING ORGANIZATION Carnegie-Mellon University 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTD)
6c. ADDRESS (City, State, and ZIP Code) Computer Science Dept Dept of Elec & Computer Engr/Dept of Statistics Pittsburgh PA 15213-3890	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center 8b. OFFICE SYMBOL (If applicable) COTD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-84-C-0063
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 63728F; PROJECT NO. 2530; TASK NO. 01; WORK UNIT ACCESSION NO. 23

1. TITLE (Include Security Classification)
RECONFIGURABLE C^2 DDP SYSTEM

12. PERSONAL AUTHOR(S) E. Douglas Jensen, Hideyuki Tokuda, John P. Lehoczky, Lui Sha, Raymond K. Clark, C. Douglass Locke, J. Duane Northcutt, Samuel Shipman, Charles Hitchcock (See Reverse)

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Apr 84 TO Oct 85	July 1988	534

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Decentralized Control, Concurrency Control, Real-Time Scheduling, Dynamic Resource Allocation, Distributed Operating System, (See Reverse)
12	07		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

ArchOS is the operating system being designed and constructed as part of the Archons research project. The goal of the project is to conduct research into the issues of decentralization in distributed computer systems at the operating system level and below. The primary research issues being investigated within Archons are decentralized control, team and consensus decision making, transaction management, probabilistic algorithms and architectural support in a real-time environment. The ArchOS computational model has been designed with both the real-time application writer and the ArchOS implementers in mind. As a result, a high level of modularity and maintainability are supported by ArchOS. The computational model features a set of cooperating instances of distributed abstract data types, called arobjects, and describes the various components of arobjects. The current set of ArchOS primitives is also described in detail to define the client's view. Finally, the rationale for our design decisions is also included.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas F. Lawrence	22b. TELEPHONE (Include Area Code) (315) 330-2158 22c. OFFICE SYMBOL RADC (COTD)

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

12. PERSONAL AUTHORS (Continued)

H. M. Brinkley Sprunt, Ragunathan Rajkumar, James Wendorf, Roli Wendorf, and Andre M. Van Tilborg

18. SUBJECT TERMS (Continued)

Failure Recovery
Best Effort

1. Summary

This report contains a summary of research at Carnegie-Mellon University (CMU) supported by the subject contract for the period 10 April 1984 through 2 October 1985. This report is prepared in compliance with CDRL Item No. A006, and in accordance with DID DI-S-3591A/T, Part 2b. The research reported herein is conducted in the context of the Archons Project, a multiply-funded effort that currently involves 25 people from CMU's departments of Computer Science, Statistics, and Electrical and Computer Engineering.

The work reported herein is divided into three major sections. Chapter 2 contains a functional description of the ArchOS decentralized operating system. Chapter 3 contains a System/Subsystem Description for ArchOS. Chapter 4 describes theoretical results in resource management that have been developed during the contract period.

DTIC
COPY
INSPECTED
6

Accesion For
NTIS
DTIC
By
Dist
Dist
A-1

2. ArchOS Functional Description

1. Executive Summary

This document constitutes the Functional Description of ArchOS (CDRL A002) for RADC Contract No. F30602-84-C-0063, Reconfigurable C^2 DDP System. This report describes the functional description of the Archons operating system (ArchOS).

ArchOS is the operating system being designed and constructed as part of the Archons research project [Jensen 84]. The goal of the project is to conduct research into the issues of decentralization in distributed computing systems at the operating system level and below. The primary research issues being investigated within Archons are decentralized control, team and consensus decision making, transaction management, probabilistic algorithms, and architectural support in a real-time environment.

It is planned that ArchOS will be developed in three stages. The first, called the interim testbed version, is now underway and will handle a small set of Sun Workstations[1] interconnected by an Ethernet. It is expected that this operating system will constitute an existence proof of many of the basic concepts involved in our research as applied to the construction of a decentralized operating system. This system will later be followed by a more complete testbed operating system incorporating the lessons learned in the interim testbed construction. Finally, an analysis of the ArchOS structure is expected to result in the construction of specialized hardware for the purpose of executing a complete ArchOS system, and ArchOS will be rewritten to run on it at that time.

This document consists of three reports, namely ArchOS Research Requirements, System Functionality Specification, and Client Interface Specification. Following this executive summary, Chapter 2 contains a description of the Research Requirements for ArchOS. The Research Requirements summarizes our high-level goals and objectives, as well as detailing some of the implications that they have on the ArchOS client interface, the structure of ArchOS, and the research activities that must be supported by ArchOS. Chapter 3 contains the System Functionality Specification for ArchOS. The System Functionality Specification identifies the Archons project' guiding concepts, such as our view of decentralized resource management and high-level view of system functionalities. Chapter 4 describes the Client Interface Specification for ArchOS, which defines the client's view of ArchOS functionalities, including the ArchOS computational model, the operating system primitives, and a rationale of the design decisons. The ArchOS computational model has been designed with both the real-time application writer and the ArchOS implementers in

[1] Sun Workstation is a trademark of Sun Microsystems, Inc.

mind; as a result, a high level of modularity and maintainability are supported by ArchOS. The computational model features a set of cooperating instances of distributed abstract data types, called arobjects, and describes the various components of arobjects. The current set of ArchOS primitives is also described in detail to define the client's view. Finally, the rationale for our design decisions is also included.

2. Research Requirements

2.1 Overview

ArchOS will be an operating system implemented as a vehicle for research into the design, construction, operation, and performance of decentralized operating systems. It will be designed to provide the complete execution environment for a single distributed program on a loosely-coupled collection of processors. For the purpose of this document, a distributed program is defined as a group of concurrent processing entities, cooperatively attempting to achieve a common goal. It is our aim to explore the contention that resource management decisions made in a highly decentralized manner will produce a highly robust, extensible, and modular system.

The development of ArchOS will take place in three steps. Initially, an interim system will be produced using off-the-shelf hardware and existing languages and support systems. This interim system is the primary focus of this set of research requirements, and will be used as a research vehicle for the development of concepts related to decentralized operating systems. Later, the lessons learned from the design and construction of this first, limited version of ArchOS will be used to produce a more complete, decentralized operating system. Finally, the interim system will be discarded and replaced by a testbed system which will incorporate the lessons learned in the implementation of the interim system, and will be implemented on a decentralized computer system designed and constructed specifically for this purpose.

2.2 Goals and Objectives

Despite the fact that the first version of ArchOS is intended to be discarded after appropriate experiments have been performed, it is the major focus of our initial decentralized operating system research and development effort. This version of ArchOS will allow us to determine the degree to which we have met our goals and objectives and will aid in the identification of areas where more study and/or support is needed.

The major goals and objectives for this phase of our research are:

- *To develop an understanding of the nature of a particularly interesting portion of the space of decentralized systems*, especially with respect to robustness, extensibility, and modularity, through the application of fundamental concepts relating to decentralized control, team and consensus decision making, probabilistic algorithms, distributed non-serializable transactions, and architectural support for operating systems.

- *To maintain high system availability* in the face of node and communications faults,

providing for no lasting degradation of system function or response beyond the actual resource loss encountered. In particular, the use of atomic transactions to maintain consistency will result in continuance of useful computation in the presence of lost, delayed, inaccurate, or incomplete information.

- *To provide and encourage high system modularity.* Previous distributed systems have been constructed around the physical communications limitations of the system, as opposed to being based on such modern software engineering principles as abstract data types and information hiding for defining modularity.

- *To construct highly extensible facilities* which will limit the cost of redesign for implementing widely divergent operating system facilities for experimentation with the fundamental concepts of interest to us in this research.

- *To provide reasonable system performance* to support applications with real-time constraints. ArchOS must consider time-critical functions for which time constraints define some system failure modes (i.e., issues of timeliness will not be ignored as they are in some systems, nor will they be dealt with by simply providing a large ratio of available to currently used resources, which is how virtually all other real-time systems strive to meet response time requirements).

2.3 Specific Requirements

The goals and objectives of the previous section are stated in very general terms. We believe that there are other, more specific, requirements that must be met with respect to the structure and services provided by ArchOS, the client's (application's) interface, and the research activities to be supported by ArchOS. Each of these areas is discussed in the following sections.

2.3.1 General Characteristics

ArchOS must provide user processes with the minimum services of an operating system (e.g., CPU management, storage management, I/O, communication management, synchronization). The level of facilities need not be uniform across the range of functions provided, but it should be tailored to the needs of the experiments to be performed. The interim ArchOS implementation specified must be implemented with off-the-shelf local area network hardware, using existing languages and an existing development environment.

All of the system resources are to be managed by ArchOS in a fashion that "best facilitates" the needs of a single program (possibly with real time constraints), that may consist of a (potentially large) number of (actually or apparently) concurrent and cooperating processing entities.

The specification, design, and (to a lesser extent) implementation of the system must not be based on any assumptions about absolute (complete and correct) knowledge of system state, consistent views of the global ordering of events, or the presence of binary error probability distributions.

The system must collect data concerning resource management decisions and instances of communication in order to provide on-the-fly or after-the-fact analysis capabilities.

2.3.2 Client (Application) Interface

ArchOS must have a client interface that allows any of the cooperating processing entities in an application to request resources in a fashion that does not require them to be aware of the actual number and location of the resources, nor of the number, location and/or resource needs of other processing entities in the system.

System security and protection are not major issues in ArchOS, and will not be handled beyond the attempt to keep one processing entity running in the event of erroneous processing by another processing entity. No attempt will be made to detect or prevent deliberate penetration. This is to say that protection in ArchOS will be defensive and not absolute.

There are no requirements for the design and implementation of the system to be concerned with application level debug or fault tolerance features. Furthermore, the client interface need not be oriented toward "user convenience" (e.g., reasonable defaults, alternative views).

The client interface is not an issue beyond the requirements described above.

2.3.3 ArchOS Structure

ArchOS should be structured as essentially identical peer modules replicated at each system node. These modules must allow the use of the "most decentralized" algorithms (that are practical) for managing several of the system's resources. These system resources should be chosen to demonstrate the applicability of decentralized resource management techniques in a variety of situations.

The degree and type of decentralization should be modifiable for experimentation with different decentralization techniques.

2.3.4 Research Activities

ArchOS must support research into the nature of a class of decentralized systems, particularly with respect to robustness, extensibility, and modularity, through the application of fundamental concepts relating to decentralized control, team and consensus decision making, probabilistic algorithms, distributed non-serializable transactions, and architectural support for operating systems.

It should also be noted that the above research activities provide the motivation for the Archons project in general, and the ArchOS operating system development in particular. That is, the Archons project is not attempting to develop a computing facility; it is attempting to perform research into the nature of decentralized resource management, especially in a real-time environment.

3. System Functionality Specification

3.1 Overview

The System Functionality Specification of ArchOS describes the Archons project guiding concepts and summarizes the ArchOS facilities. In this chapter, we will focus on the higher level guiding concepts which have acted as a basis for ArchOS development and summarize the system functionalities and new concepts. In the following chapter, namely in the Client Interface Specification, we will provide the detailed description of the system functionalities, including ArchOS' computational model and all of the ArchOS primitives.

3.2 Archons Guiding Concepts

In this section, we first describe the objectives of the Archons project and ArchOS. Then, following the detailed explanation of our view of decentralized resource management, an overview of the ArchOS system facilities will be provided.

3.2.1 Archons and ArchOS Objectives

The objectives of the Archons project in general, and of its ArchOS operating system portion in particular, differ significantly in a number of ways from those of the other distributed system and distributed/network operating system efforts of which we are aware

The foremost of these dissimilarities has to do with our concentration on the special case of "distribution" which we term *decentralization* (explained further in Subsection 3.2.2):

- to explore the fundamental nature of making and carrying out decisions in a highly decentralized fashion
    - o for resource management in general,
    - o but for operating systems in particular;
- and thereby to facilitate the creation of
    - o substantively improved computer systems in general (including uniprocessors and computer networks),
    - o but especially a novel *decentralized computer* which can be physically dispersed yet which exhibits the optimality of executive level global resource management hitherto confined to physically concentrated (and highly centralized) uni- and multi processor computers.

Our principles of decentralized decision making and resource management have wide applicability, from integrated man-machine systems, through application software, operating systems, and down to machine hardware. However, we are focusing on the OS levels (and below) for three important reasons:

- the OS is a constant beneath many changing applications —

  this provides generality and lowers system costs by solving resource management problems once instead of leaving them to be solved repeatedly by the users;

- the degree and cost of successful decentralization above the OS depends on success at the OS levels and below;

- OS problems are almost always the most general, complex, and dynamic (elsewhere in a system the resources are usually more dedicated) —

  consequently, solutions at the OS levels are more likely to be amenable for use at higher or lower levels, while the converse is much less likely.

Any system can be expected to have resources local to each node, but we are disregarding them in our research since managing them is so well understood by comparison with managing global resources.

We review the most salient aspects of our quest for decentralized global executive resource management in the next subsection. Following that, we list some various other distinguishing facets of our venture.

3.2.2 Decentralized Resource Management

There currently seems to be little common understanding about what "distributed" decision making or resource management means; this is one of the reasons that the "distributed" and network operating systems in the literature and laboratories are so very conceptually and functionally disparate. We have chosen to use the term "decentralization," and to attempt to rather carefully (albeit not formally) define what we mean by it.

"Centralized" and "decentralized" are not usefully viewed as a dichotomy, but rather as the endpoints of a continuum -- indeed, as diagonally opposed vertices of a multidimensional space. We are not under a misconception that extreme decentralization is necessarily advantageous in all ways and under all circumstances. However, our experience (both prior to and subsequent to the initiation of the Archons and ArchOS research) provides cogent arguments and concrete evidence that movement away from the heavily populated highly centralized subspaces can be invaluable. At least

in some applications we are familiar with, such as supervisory real-time control (e.g., combat platform management and factory automation), more decentralization of resource management offers improvements in certain system attributes like robustness and modularity. In order that we, or any designers of a particular system, be able to *scientifically* position ourselves well in this space from maximally centralized to maximally decentralized, far too little knowledge exists today about its decentralized boundary conditions.

It became apparent to us that these issues ought to be dealt with explicitly and systematically from the ground up, not in the prevalent ad hoc adaptive (albeit safer and faster) way, if the many attractive promises of a physically dispersed computer were to be realized. Thus, we launched an extensive search for the limits of resource management decentralization, divided into two areas: logical and physical. (Computer scientists sometimes imagine incorrectly that logical things are innately more conceptually interesting than are physical things; the opposite seems true to us in this case.)

3.2.2.1 Logically Decentralized Resource Management

One of our first steps was to create a conceptual model of the space of *logical* decentralization of decision making; the most detailed of its incarnations can be found in [Jensen 81a]. It can be applied at different levels of abstraction, from one instance of one decision about one resource, through all instances of all decisions about all resources. Our model is germane to the management of local or global resources. In it, decentralization is founded on *multilateral* management; not, for instance, on the more common theme of resource or functional partitioning (which leads to autonomy as maximally decentralized, which we reject). Our model expresses the degree of decentralization as being determined by several factors, crudely summarized as follows:

- the percentage of resources involved;
- the percentage of decision makers which participate (depending on the level of abstraction under consideration, functional partitioning and successive techniques such as round robin may be placed at the centralized end of this axis);
- the extent to which all decision makers must become involved before a decision has been completed (note that resource partitioning and functional specialization are highly centralized by this metric);
- the degree of equality of decision maker authority and responsibility (this axis places a premium on peer relationships, in contrast with the ubiquitous hierarchical ones).

To this version we subsequently added a negotiation axis.

At the minimum, more centralized, end of the negotiation axis, each decision is made by a collection

of entities which work as a "team" to move the overall system toward its goals. Any team member is allowed to make certain (not necessarily fixed) decisions without necessarily gaining the concurrence of the other team members; these decisions may be constituent subdecisions or different instantiations of the same decision (this difference is represented on other axes of the model). Any team member may seek information which will improve the quality of its decisions [Marschak 72]. A well-known example of team decision making is routing in the ARPANET communication subnet [Ahuja 82].

At the opposite, more decentralized endpoint, each member in a collection of decision makers develops hypotheses (deductions and assumptions) with associated probabilities — these may be based on some form of partitioned competence (designated elsewhere in the model) or disparity of information (which may again be a logical factor, or a physical one as discussed in the following subsection). To make a decision, members exchange these, reason about and modify them, making compromises as necessary, and in this way enhance the marginal viewpoints to a more global view. This activity must somehow converge to a single consensus decision, perhaps by a formal method such as that of DeGroot [DeGroot 74], or by heuristics such as inference rules and algorithms — the latter are employed by some areas of artificial intelligence such as problem solving and expert systems (e.g., the HearSay system [Erman 73], [Erman 79]). (Note, however, that HearSay was quite centralized by our standards: logically, because the knowledge sources were functionally specialized; and physically, due to the shared global state "blackboard".) This technique is somewhat similar in spirit to the "divide and conquer" approach of algorithm design but lacks the optimality of the full Bayesian method, because the joint information has been sacrificed. That is, the various inter-dependencies are only approximated by the indirect approach of reaching a consensus.

We are interested in the conditions under which different degrees of logical decentralization according to our model offer how much of which attributes, and the tradeoffs involved, in managing global resources. But the region of primary interest to us in this multidimensional space of logical decentralization is where each global decision is made multilaterally by a group of peers through negotiation, compromise, and consensus.

According to our view, most resource management is highly logically centralized, even in the myriad network and distributed operating systems we are aware of.

3.2.2.2 Physically Decentralized Resource Management

We have long argued that the important benefits of having *system-wide* resource management at the *operating system* level, routinely provided by a computer, are not available to many systems — the reason is that those systems consist of multiple nodes which must be physically dispersed (for functionality, reliability, and logistical reasons).

Unfortunately, operating systems as presently conceived are highly and inherently centralized in several critical respects. Perhaps most importantly, they are based on some very strong premises about time — e.g., that communication delays due to physical dispersal within the operating system are practically negligible with respect to the rate at which the system state changes (note that the same effect can occur on a single VLSIC/VHSIC chip). This leads to the presumption that it is possible (and even cost-effective) for all processes to share as complete and coherent a view of the entire system state as may be desired (e.g., that a single global ordering of events can be established). Another class of centralized operating system premises has to do with the types, frequencies, and effects of faults, errors, and failures. Both the time and fault premises are rational given the historical evolution of operating systems in the context of shared primary memory (i.e., uniprocessors and multiprocessors). Unfortunately, many of these premises go unstated (e.g., in operating system texts and papers), and are either forgotten or assumed to unquestionably always hold.

Our focus is on achieving the global executive level resource management for a physically dispersed system to be a *computer* in the same sense that a uniprocessor or multiprocessor is. However, we are not restricting ourselves to virtual uniprocessors — is it frequently beneficial (e.g., improved fault recovery and performance) for some image of the composite and decentralized structure of the software or hardware to (occasionally or optionally) be made or left visible to the user.

Presently, Archons appears to be essentially alone in stressing unification at the operating system levels; the dominant theme in distributed system projects today is "autonomy." The only popular alternative to conventional centralized computers is *computer networks*. A conventional generic computer network can be characterized as follows:

- each computer is (functionally and often administratively) autonomous with its own local, centralized operating system;
- all the computers are connected to a communications subnetwork;
- each computer has network server utility software (for transport protocols, naming conventions, and the like) sufficient for them to do resource sharing (e.g., file transfers, mail, virtual terminals);

- there may be higher layers of software for specific applications (e.g., banking, military C^3), perhaps giving the users some unified perception of the system.

A network normally is supplied with a so-called "network operating system", which tends to simply be the collection of network server utilities. Historically it has been constrainted to being a guest of the local operating systems; recently, more indigenous (and thus more effective) network operating systems are developing (e.g., [Rashid 81]). A few recent networks and their network operating systems aspire to eventually make gradual movement in the direction of greater operating system coordination (e.g., [Spice 79]).

Many applications need nothing more than long-haul resource-sharing or value-added networks, or local area networks of personal workstations. But not in the cases of concern to us: the very complex problems of achieving system-wide resource management are forced up to the user level where they are more difficult (having less access to lower level resources and receiving little assistance and perhaps even resistance from the local operating systems); and where they must be solved repeatedly if there are multiple users, instead of once by the system designers. The unsurprising consequence is that these applications with substantive state change/visibility ratios must suffer: because of the solopsistic local operating systems, *system* robustness is poor, modularity is compromised, performance (e.g., concurrency) is reduced, and total system cost is increased.

We fought with the dilemmas of this dichotomy between computers and computer networks during the past decade of our experience designing distributed systems and realized that having a physically dispersed computer requires a functionally singular operating system (as opposed to a network of independent private operating systems). The primary obstacles to be overcome are that:

- communication *within the operating system* is inaccurate and incomplete with respect to system state changes;

- and the types and effects of faults, errors, and failures encountered in multinode physically dispersed systems differ significantly *in both degree and kind* from those in single node systems — this is even more pronounced in a decentralized computer than in a computer network.

3.2.2.2.1 Accommodating Imperfect Information

High degrees of physical decentralization imply that resource management decisions routinely must be "best effort", based on imperfect quantity and quality of information — the virtually ubiquitous "garbage in, garbage out" characterization of computers is unrealistic and cannot be tolerated in a physically dispersed multinode computer. This perspective is somewhat familiar *above* the OS levels (e.g., in certain artificial intelligence work), and *below* them (e.g., in dynamic communication packet

routing). But *at* the OS levels (as in most software), it is a foreign outlook which is incompatible with the current state of the art. Consequently, new problems have to be solved in the design of the decision algorithms, such as: picking thresholds of result acceptability, and specifying them to the decision makers; determining what "value" the completeness and accuracy of information utilized contributes to the "quality" of a decision result.

In thinking about these issues, it becomes clear that while the logical and physical aspects of decentralized decision making are conceptually distinct, they strongly interact. For example, the decision convergence time may include acquiring suitably valuable quantity and quality of information, as well as negotiating.

An unavoidable characteristic of a physically dispersed machine is a significant increase in the indeterminism of its behavior. The centralized mind set is that not only ends but also means at all levels ought to be entirely deterministic; it is normally affordable to closely approximate this in a centralized machine, and instances to the contrary are dealt with in various ad hoc fashions. Our position is that considerable indeterminism is the *normal* case in decentralized resource management, and can be exploited to advantage (e.g., improved robustness and performance) rather than merely tolerated. Dynamic packet routing demonstrates our position, and we have done so (transparently to the users) in a network operating system [Sha 83a].

3.2.2.2.2 Faults, Errors, and Failures

In a physically dispersed multinode system (whether network or computer), reliability problems are worse than in nondispersed and uninode systems, particularly when considered in light of the imperfect information issues discussed above. For example, concurrency control and failure atomicity become vastly more complicated, far beyond the realm of current centralized operating system conceptions. Computer networks have more relevant technology in this respect, but most of it is actually inspirational rather than directly transferable to a decentralized OS and computer.

We address failure management and recovery within an operating system by thinking of the OS state as a special kind of distributed database which is approximately replicated at each node. This suggests that an atomic transaction facility for use by the OS (and perhaps by higher, e.g., application, levels) be incorporated in each instance of its kernel ([Jensen 81b], although we made this pivotal design decision in 1978 as a result of several enlightening discussions with Gerard Le Lann about his distributed database concurrency control research). As a consequence, three classes of significant research issues arise.

One is that both the services and structure of the OS ought to be substantively affected by the availability of atomic transactions as kernel primitives. This is essentially virgin territory: the few approximations to atomic transactions at the OS level have been ad hoc; in fact, they have not been explicitly viewed, designed, and exploited as transactions.

Secondly, the overhead (especially communication) of atomic transactions, which is always a concern, becomes of paramount importance at the OS kernel level. We have determined that one can achieve great acceleration without degrading flexibility with thoughtfully crafted hardware *mechanisms* at the disposal of software modules which establish the desired policies.

The third class of research issues has to do with the need for insightful reconsideration of atomicity itself. While the inclusion of atomic transactions in our OS was inspired by their contributions to conventional database systems, our transaction facility differs radically in several respects from those used in that context. Examples include the following.

The conventional serializability theory of concurrency control includes the assumption that consistency and correctness result from a single transaction executing alone; this leads directly to the same result for all serializable schedules. An advantage of serializability is that it is completely general and works without requiring knowledge about either the database or the transactions; but a disadvantage is that it cannot exploit such knowledge which may be available in any specific case (such as in an OS). The cost of this generality is that serializability can exclude consistent and correct schedules which provide higher concurrency than those it permits.

Database researchers have begun to study *nonserializable* consistency control methods in search of greater concurrency, but a major difference is that a transaction can no longer be regarded as if it were executing alone. Consequently, the consistency and correctness properties of nonserializable scheduling rules do not follow automatically, they must be proven. So that every attempt to utilize nonserializability is not burdened with inventing its own rules and proving their properties, a formal theory of nonserialzable concurrency control is needed. Because it is already known that serializability theory provides the highest degree of concurrency possible when using only the classically defined transaction syntax, some additional kind of information is required if nonserializabilty is to perform better.

Researchers other than ourselves seem to be focused exclusively on exploitation of transaction *semantics*, developing syntactic structures to support programmers' specification of their own application dependent scheduling rules. All programmers involved with any given database must

understand the details of each other's transactions, and every programmer is responsible for the consistency and correctness properties of his own rules. Such extensive use of global transaction semantics (e.g., the "break point specifications" in [Lynch 83] and "lock compatibility tables" in [Schwarz 82]) allows very high degrees of concurrency, but appears to limit this approach to rather static and specific situations.

Modularity is recognized to be extremely valuable in software engineering generally; we consider it a critical attribute in transaction-based distributed computations, particularly decentralized operating systems. Therefore, we have created a different and more decentralized theory of nonserializable concurrency control which seems improved with respect to modularity; when a programmer schedules one of his transactions, he need know only the agreed upon transaction syntax, the details of his own transaction, and the consistency constraints of the database subset affected by this transaction. We define a new transaction syntax, called *compound transactions* (of which nested transactions are a special case), and its associated *generalized setwise serializable* scheduling rules (of which serializability is a special case). Our schedules are *complete* in the sense that for any consistency and correctness preserving schedule, there exists an equally consistent and correct setwise serializable schedule which provides at least as much concurrency.

An important implication of our different approach to transactions is that failure management and recovery must be re-evaluated. The usual notion of failure atomicity, commonly thought to be an end, is in fact a means to maintaining consistency and correctness in the event of failures when using serializability theory, where a transaction cannot be commited until all its actions are successful and in stable storage. Higher concurrency can be achieved by determining conditions which permit a transaction to commit completed steps before the end of the transaction. Such conditions are a natural consequence of generalized setwise serializable transactions, enabling us to introduce the concept of *failure safety*. In [Sha 84], we formalize our theory, and its properties of consistency, correctness, modularity, optimality, completeness, and failure safety.

Other major departures we must make from normal atomic transaction facilities include:

- Instead of being above an OS with the corresponding functionality to draw on, our transaction facility is *beneath*, inside the OS kernel. This affects the facility design substantially.

- Rather than handling simple database objects such as records and files, it must accommodate the far more complex, abstract, and dynamic data types found in an OS.

- An object is not necessarily located at a single node --- a single instance of it may be physically dispersed [illegible] multiple nodes.

Note that the work outlined above has applicability beyond our motivation to enable the creation of a logically and physically decentralized operating system (and computer) which is extremely reliable and modular.

3.3 ArchOS Facilities

The ArchOS system is intended to be a vehicle with which our research issues can be investigated, rather than an operational applications programming environment. Thus, the ArchOS facilities must have sufficient functionalities to allow test applications to be constructed with which to study ArchOS' characteristics, but they need not provide a particularly complete set of facilities. The set of facilities provided, then, will be evaluated with respect to their support of Archons' primary research goals, rather than with respect to an operationally complete functionality.

Nevertheless, the set of functions described in the Client Interface Specification (i.e., Chapter 4) should describe a set of facilities each of which is functionally closed: i.e., each function provided is complete so that its use can be measured with respect to Archons' research issues. Thus, for example, in the handling of transactions at the client interface, all of the important issues in transaction management are covered, even though other functions, such as file management, may be missing or skimmed. It is not to be assumed that missing or incomplete functionality in the interim tesbed ArchOS represents valueless functionality, but rather that the issues involved with such functions are not within the primary Archons research interests. Functionality in such areas is expected to be much more complete in later versions of ArchOS incorporating the results of the research performed during this early ArchOS implementation.

In this section, we describe the important ArchOS facilities focusing on our design decisions and their relationships with Archons' research issues.

3.3.1 Arobject/Process Management Facilities

The arobject and process management facility provides the basic functions to create, remove, bind, or unbind cooperating processing entities in ArchOS. ArchOS provides completely network-transparent arobject/process management. That is, creation and destruction of arobjects and processes can be performed without knowing the location of the target arobject or process.

We view an arobject as a basic module for embodying a distributed abstract data type. In particular, we decided to treat an arobject as an *active* system entity rather than a *passive* entity. The major reason is that an arobject can be a very natural building block for [illegible] of distributed programs and

for construction and organization of cooperating resource managers. In particular, there are several advantages related to the reliability of the system. First, we can easily define an autonomous module. It is easy for an arobject not only to initialize its computational state by itself, but also to recover its computational state by itself. In other words, by having a single INITIAL process in each arobject, a designer can give responsibility for recovery to the INITIAL process. Second, unlike traditional procedure invocation in abstract data types, a caller cannot invoke an operation of an arobject in a *master-slave* manner, but must use a form of rendezvous. The receiver therefore has the right to accept, reject, or delay the requested function. Finally, the degree of parallelism within an arobject can be dynamically changed in many ways. For instance, a process can be created in a different node to perform a requested computation on demand or many processes can be pre-created to accept a particular type of request.

3.3.2 Communication Facilities

ArchOS communication facilities provide support not only for a conventional client-(single)server model but also for cooperation among multiple servers in a distributed environment. Thus, the system supports a *Request-Accept-Reply* type of communication among cooperating arobjects in either a synchronous or an asynchronous manner.

To support such a cooperating server model which will utilize our notion of distributed best effort decision making[2], we also created a *one to many* type of communication among cooperating arobjects. That is, a process can invoke an operation on a group of particular arobject instances at once and does not need to block itself until one or more replies to come back. In this way, the requestor can also perform its own activity while the servers are working on the requested operation.

As for various server structures, we are interested in a collection of servers which can increase the availability and reliability of service. The productivity of the servers may also increase together in a collection of cooperating workers. Even for the single server case, it is easy to make an arobject autonomous and cooperative, since there is at least one process managing its service. Unlike the master-slave relationship in remote procedure calls, the server can control not only the sequence of incoming messages, but also the execution order of the actual services.

Another important characteristic of the communication facility is that each communication primitive is handled as, so called a compound transaction (see Section 3.3.3) by ArchOS. Thus, the system

---

[2] We cannot define the best effort decision making in precise manner at this point. However, we view that it refers to the capability for decision making utilizing incomplete and inconsistent information in the best possible way.

can guarantee the atomic property of each communication activity. In other words, the system performs either all of the message communication activities completely or nothing at all.

3.3.3 Transaction Facility

The ArchOS transaction facility provides two types of transactions, elementary transactions and compound transactions, which may be nested in arbitrary combinations. Elementary transactions correspond to the traditional nested transaction facility [Moss 81] if only elementary transactions are nested. Compound transactions [Sha 84] provide more potential concurrency than elementary transactions and differ from elementary transactions in that when a compound transaction commits the effects of the transaction commit processing takes place immediately.

Unlike a traditional database transaction facilities, ArchOS itself will be built by using compound and elementary transactions. These OS-level transactions should facilitate the maintenance of system consistency while simplifying data sharing among multiple processes. Another different characteristic is that the ArchOS transaction facility does not provide failure atomicity and permanence for all of the data accessed by a transaction. Rather, only the data items that have been explicitly declared to be *atomic* have these properties.

We have attempted to view ArchOS as a highly decentralized, real-time application built on top of the kernel support facilities at various times, and that has led us to view the arobject as a computational entity that we would use *within* ArchOS (insofar as possible), as well as at the application level. While specifying the services to be provided by ArchOS and its behavior under all conditions, it became apparent that ArchOS' internal system data objects should possess several characteristics. In particular, the following characteristics were desired:

- Often, several different arobjects will operate on specific data items (directories, queues, and so on). These shared data objects will reside in an arobject, so access can be coordinated by the normal arobject communication facilities (particularly *Accept* primitives), as well as by means of customized code in the arobject's processes. But, as the following points illustrate, a higher level coordination will often be desirable.

- At times, it is necessary to change several different, yet related, data items as a unit (for example, updating all of the copies of an entry in a partially replicated directory or moving an element from one queue to another). If such atomic updates could be performed, then it would be much easier to transform one consistent state of a set of data items to another consistent state.

- Some data items must be permanent; that is, their state should be reliably maintained for the life of the system.

These attributes could all be provided by ArchOS by means of the communication facilities in conjunction with custom-written code and appropriately defined locks or semaphores and critical regions. However, we desired a more structured approach. All of the above capabilities are supported by traditional database transaction systems [Gray 77]. In fact, such transaction systems can provide even more powerful properties (specifically, failure atomicity and/or serializability). It was felt that this additional structure would ease the programmer's burden, while also decreasing the chances for programming errors, by making modular programming more natural. Indeed, the use of compound transactions promotes modular construction of programs by causing the transaction author to think in terms of consistency preserving transformations on sets of atomic data objects, thereby causing the program's data to be partitioned into a number of modular atomic data sets. Also, transaction systems in general can aid the programmer in another important way: the processing carried out in order to commit or abort a transaction can handle a great deal of lock-related bookkeeping, even though the programmer must explicitly obtain locks on all of the atomic data items. This frees the programmer to consider the correct behavior of the transactions being written, without giving unnecessary consideration to interactions with other transactions in the system. However, this is not to say that the programmer does not have to be concerned at all with locks and locking protocols; rather, it is intended to point out one aspect of lock management that the programmer does not have to handle. Using the current ArchOS locking protocols, there are still a number of decisions concerning locks that the programmer must make--for instance, whether to use a discrete locking protocol or a tree locking protocol, how to organize the locks in a lock tree, what data items are associated with a given lock, and so on.

3.3.4 File System

The current file management in ArchOS supports a single-level file structure and does not provide any directory structure for users. However, the system can provide at least three kinds of file properties. First, a *normal* file has the data portion of the file arobject in volatile storage. Second, a *permanent* file's data portion is allocated in permanent (non-volatile) storage. Finally, an *atomic* file has the data portion which is declared as an atomic data object.

Although the current file system is a single level file structure, a client does not need to know the exact location of the file itself. Furthermore, since a file is designed as an instace of "FILE" arobject, a client will be able to create a new type of service, such as a directory, a replicated atomic file type, access list, etc., on top of that.

3.3.5 Real-Time Facility

The real-time facility of ArchOS provides essential functions to perform time-driven scheduling such as obtaining time information and setting/resetting scheduling parameters for a client program. This time-driven scheduling is based on ArchOS' "best effort" scheduling model.

Since the application writer best knows the real-time constraints for a task, the ArchOS client program should be able to define application-dependent deadlines by which time some specific processing milestone must be reached. ArchOS should take this deadline information, along with any other relevant parameters and schedule the runable processes in a manner that best meets the scheduling policy requirement defined by the client.

The client may specify scheduling policies that define system behavior in the case where the system is overloaded. For instance, the client can specify that the scheduler should attempt to minimize the number of missed deadlines or minimize the average lateness.

By coordinating with a scheduling policy, a client may also create an application-dependent reconfiguration policy. The scheduling and reconfiguration algorithms, under control of the appropriate policies, will be able to determine a (sub)set of processes which should be removed from the local node.

3.3.6 Policy Management

The separation of policy and mechanism in ArchOS is important to meeting its flexibility requirements. Some of the system facilities, such as process scheduling, require the specification of policy with respect to the management of the resources involved, using mechanisms defined within ArchOS. Thus, an application designer of ArchOS is able to create a tailored real-time environment by appling his/her own best effort decision making.

For example, *Schedule* and *Reconfigure* will be recognized by ArchOS as *policy names*. All of the client defined policies are maintained by a *policy set arobject* which will be instantiated by the application program. Thus, a client can add, delete, or change an existing policy by invoking a corresponding system primitive during run time.

3.3.7 System Monitoring and Debugging Facilities

The system monitoring and debugging facilities provide various abilities to control the behavior of cooperating arobjects and their processes during execution. For instance, a client process can freeze or unfreeze a target arobject or process's activity and can watch or capture incoming or outgoing messages of the target. However, the system monitoring and debugging facilities are not offered to a normal client arobject, rather to a specially privileged arobject in the system.

The following functionalities will be created on top of the current monitoring and debugging facility.

- Single-step function for tracing process activity
- Creating an arbitrary break point in an arobject
- Specialized debug functions such as *Redo* or *Undo* operation for arbitrary functions

4. Client Interface Specification

4.1 Overview

ArchOS is the operating system being designed and constructed as part of the Archons research project. The stated goals of the project are to conduct research into the issues of decentralization in distributed computing systems at the operating system level and below. In this area, the primary research issues being investigated within Archons are decentralized control, team decision making, transaction management, probabilistic algorithms, and architectural support.

It is planned that ArchOS will be developed in three stages. The first, called the interim testbed version, is now underway and will handle a small set of Sun workstations[3] interconnected by an Ethernet. It is expected that this operating system will constitute an existence proof of many of the basic concepts involved in our research as applied to the construction of a decentralized operating system. This system will later be followed by a more complete testbed operating system incorporating the lessons learned in the interim testbed construction. Finally, an analysis of the ArchOS structure is expected to result in the construction of specialized hardware for the purpose of executing a full-blown ArchOS system, and ArchOS will be rewritten to run on it at that time.

In this document, we describe the characteristics of the interim testbed version of ArchOS currently being designed. This system is intended to be a vehicle with which these issues can be investigated, rather than an operational applications programming environment. The client interface, then, must be have sufficient functional completeness to allow test applications to be constructed with which to study ArchOS' characteristics, but need not provide a particularly complete set of facilities. The set of facilities provided, then, will be evaluated with respect to their support of Archons' primary research goals, rather than with respect to an operationally complete functionality.

Nevertheless, the set of functions described in this document can, and should, describe a set of facilities each of which is functionally closed: i.e. each function provided is complete so that its use can be measured with respect to Archons' research issues. Thus, for example, in the handling of transactions at the client interface, all of the important issues in transaction management are covered, even though other functions, such as file management, may be missing or skimmed. It is not to be assumed that missing or incomplete functionality in the interim tested ArchOS represents valueless functionality, but rather that the issues involved with such functions are not within the

---

[3] Sun Workstation is a trademark of Sun Microsystems, Inc.

primary Archons research interests. Functionality in such areas is expected to be much more complete in later versions of ArchOS incorporating the results of the research performed during this early ArchOS implementation.

This document consists of four chapters. Following this introductory chapter, chapter two will contain a description of the computational model of ArchOS. This will include the client's view of application software structure, with a description of the primary ArchOS client facilities, language issues and the general application software development environment as supported by ArchOS. Chapter three will describe the operating system primitives, including a brief high level description of the underlying operating system activities likely to be undertaken in response to each of the primitives. Particular attention will be given to ArchOS' handling of resource management and consistency, as well as recovery issues. Chapter four will provide a rationale of the design decisions described in Chapters two and three, outlining the tradeoffs made in the selection of application structures and the semantics of the operating system primitives.

4.2 ArchOS Computational Model

In this model, we mix two paradigms in the distributed domain: *object-oriented* and *process-based* programming. While this mixture is hardly novel (e.g., [Lazowska 81]), our paradigm differs from others in several ways. The primary goal is a high level of modularity and maintainability for both the real-time application implementer using ArchOS, and for the ArchOS implementers themselves.

4.2.1 Principal Components

The purpose of ArchOS is to provide an execution environment for a real-time decentralized system, such as a command and control system. The application software of such a system can be considered to be a single distributed program. The distributed program can be described in terms of its primary constituent components, which can be further broken down until the familiar sequential components (processes) are described.

4.2.1.1 Distributed Program

A distributed program consists of one or more arobjects (major program modules -- see Section 4.2.1.2) working toward a single goal. Generally, in a real-time system, the entire system can be considered to be executing a single such distributed program. It is possible, however, that more than one distributed program could be running in a system simultaneously (particularly during system test activities), subject to the availability of resources, but the presence of more than one program may render certain performance specifications untenable because of the associated arbitrarily resolved

resource conflicts. Each program contains a single *Root* arobject which will, in turn, spawn other arobjects.

In fact, because ArchOS will be a testbed operating system for the foreseeable future, it is expected to be used almost exclusively in "system test activities". Because of this, we have planned the client interface to allow more than one program to be executed simultaneously. While it is true that performance specifications may be compromised during such tests, the majority of such tests may not be greatly impacted. ArchOS will keep track internally of which program components are part of each separate program, using a fixed set of policies for handling resource allocation conflicts among the programs. The remainder of this document, however, will concern itself with the handling of a single program.

4.2.1.2 Arobject

An arobject is a distributed abstract data type consisting of two principle parts: a *specification*, and a *body*. An *instance* of an arobject can be dynamically created using the *CreateArobject* primitive. More than one instance of a single arobject can be created, each having a distinct identity, forming an arobject *class*. The arobject identifier returned by the create primitive identifies the particular arobject instance to be addressed during communications, or arobject communications may be initiated to an entire arobject class, using existential or universal quantifiers to specify destination addressing (see description of *Request* primitive in Section 4.3.3.1).

The lifetime of the arobject is under user control and is potentially unlimited. The system build procedures place the uninstantiated copies of the compiled and linked arobjects on long term storage (e.g. disk), but the creation of an arobject causes an instance of the arobject to be created and uniquely named. An arobject instance will be removed at the end of its lifetime, or a kill operation can remove an arobject instance.

4.2.1.2.1 Arobject Specification

The arobject specification, describing the external user's view of the arobject, consists of a set of data types and a set of operations which other arobjects will use to activate services offered by the arobject. The specification, although it completely specifies the external interface to the arobject, makes no commitment with respect to the number of processes within the arobject, their functions, or their distribution. The operations are specified in a manner similar to functions in procedural programming languages: the operation is specified with its name, its input arguments, and its output arguments. All operation invocations and replies use call-by-value semantics.

It is important to note that there is no defined relationship between an arobject's operations and the

entities (processes or procedures) in its body. Any operation could, in principle, be handled by any process in the arobject (see description of the *Accept* primitive in Section 4.3.3.4).

4.2.1.2.2 Arobject Body

The arobject *body* consists of descriptions of private data types, private abstract data types, private operations, private arobjects, and processes. Every arobject body must contain at least one process, but the other components of an arobject body are optional. This private information is visible only to processes within the arobject. If a process is not resident in the same node as an instance of a private abstract data type, the semantics of a call to one of that private abstract data type's procedures are those of a remote procedure call. The semantics of the remote procedure call will ensure that the procedure will be called at most once; it will be the responsibility of the calling process to handle the condition of a procedure never being executed.

Thus, there are a number of potential items in the arobject body, each with a set of semantics controlling their use. The procedures defined within a private abstract data type will determine the access rules to the encapsulated private data, handling mutual exclusion if needed, or providing "dirty" access if this is acceptable to the client system functional specifications. The private data can be any normal data types, but may also be defined as *atomic* or *permanent*. Data defined as atomic will be forced to stable storage upon commitment of a transaction, or restored to its prior, or some equivalent, state upon transaction abort. Permanent data is similar, except that the copying of this data to stable storage is done asynchronously (with respect to the changes made to the data) by the operating system or by an explicit primitive, with no guarantees with respect to consistency. Atomic data access is possible only within transactions (see Section 4.2.4).

Private operations and private data types are exactly the same as their counterparts in the specification part of an arobject, except that their existence is invisible outside the arobject; hence these operations and types may be used only within the arobject. Such operations and types serve to enhance the modularity characteristics of arobjects by providing for nesting of distributed abstract data types. Similarly, private arobjects are simply arobjects which, once instantiated (see *CreateArobject* primitive, Section 4.3.2.1) by an arobject, are visible only to the processes within the outer arobject. Their existence is unknown to external arobjects, and can be used to hide implementation details in exactly the same way as normal abstract data types.

4.2.1.3 Processes

A process is a sequential execution unit within an arobject; it is the dispatchable entity as viewed by the operating system. An arobject contains one or more processes, of which one may be named INITIAL. If such a process exists, it is automatically invoked at arobject creation time (see description of *CreateArobject* primitive in Section 4.3.2.1), providing for initialization of the private variables which comprise the arobject's state. Processes may share a data object with other processes in an arobject by encapsulating it as an instance of a private abstract type. Processes may be explicitly created only by other processes in an arobject: their existence is invisible to processes outside the arobject. Once created, processes are terminated at their own request, at the request of any other process in the arobject, or at the termination of the arobject instance.

Processes are *lightweight*; i.e., no computational state (such as local data) is implicitly transferred to a process via invocation other than its formal input parameters, so the invocation activity can be made to be procedurally inexpensive, particularly if appropriate hardware support is available [Jensen 84]. There is no necessity that all processes in an arobject be located at a single node. The model does not even require that processes sharing variables be located at a single node, but the actual implementation might contain such a limitation.

4.2.1.4 Sample Arobject

The structure of a distributed program and each component of an arobject is illustrated in Figure 4-1. The three small boxes in the left portion of the figure represent three cooperating arobjects. Arobject 2 has made a *Request* of arobject 1, and arobject 1 has sent a corresponding *Reply*. Similarly, arobject 3 has requested some service from arobject 2 and received a reply.

While the small boxes that represent arobjects in Figure 4-1 hint at the internal structure of an arobject (they show that an arobject has two parts and that one part contains internal processes and private abstract data types (padt's)), they do not show much detail. The large box on the right side of the figure is an exploded view of arobject 2. It shows that the two parts of an arobject are a specification and a body, and it shows the various components of each part. (These components were all discussed earlier in this chapter.)

4.2.2 Communication Facilities

Arobjects communicate via invocation parameters and messages. Unlike a remote procedure call (RPC) mechanism, the requestor and server must agree to communicate with each other. Their relationship is thus a symmetrical pair of cooperating arobjects rather than that of a master/slave. Such a symmetrical pair can be used to produce either client/server systems, or cooperating arobject

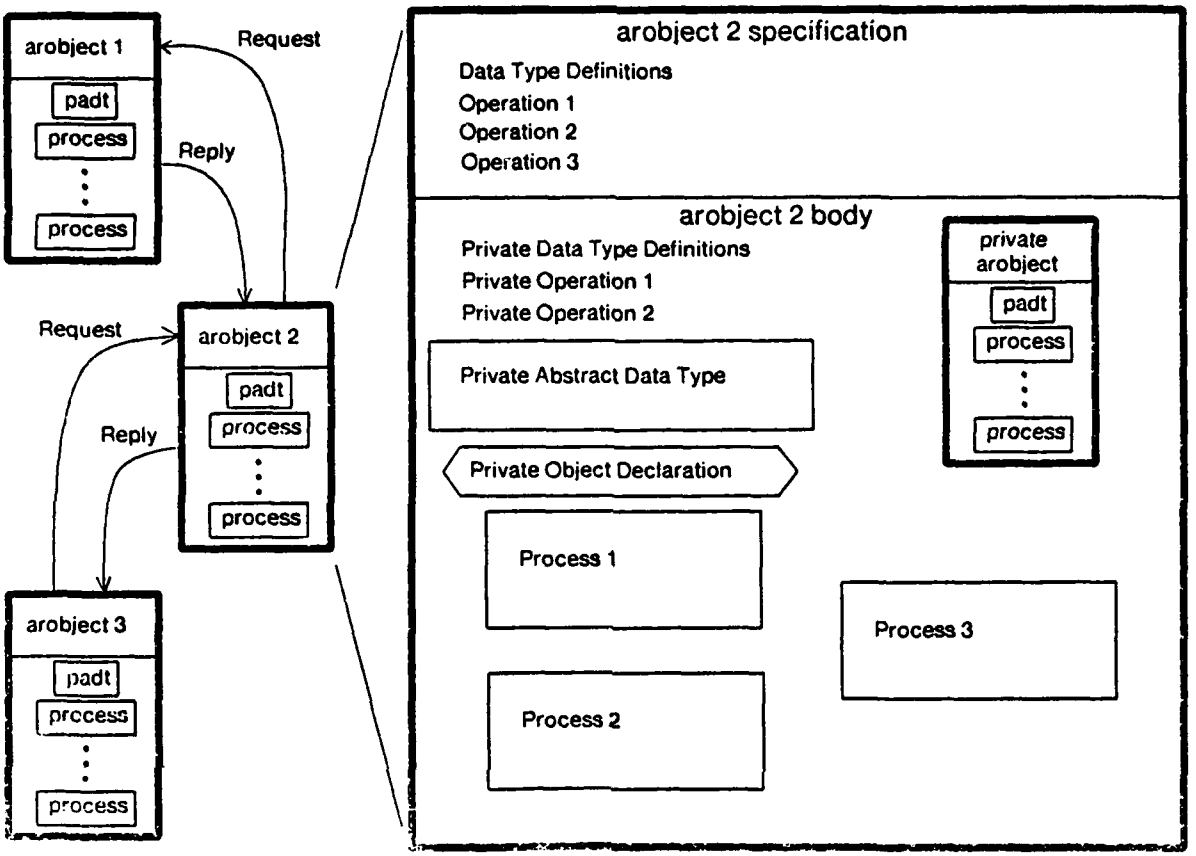


Figure 4-1: Distributed Program using Arobjects

systems, or more likely, a mixture of both. This view pervades both the application and ArchOS implementation paradigms being constructed.

In this example shown in Figure 4-2, the requestor sends a request message explicitly via the *Request* primitive and the server (i.e., Process A) performs an *Accept* primitive. The *Trans-Id* variable in the requestor is set to the unique request transaction identification if the request initiation is successful, and is set to a null transaction id ("NULL-TID") if an error has been detected (e.g., invalid arobject id). The body of the request message must match the corresponding operation parameter template provided in the target arobject specification, using call-by-value semantics. Similarly, the body of the reply message contains the operation's return-parameters. If a transaction is in progress when the request is made, ArchOS will manage the applicable transaction semantics, depending on the transaction type (See Section 4.2.4).

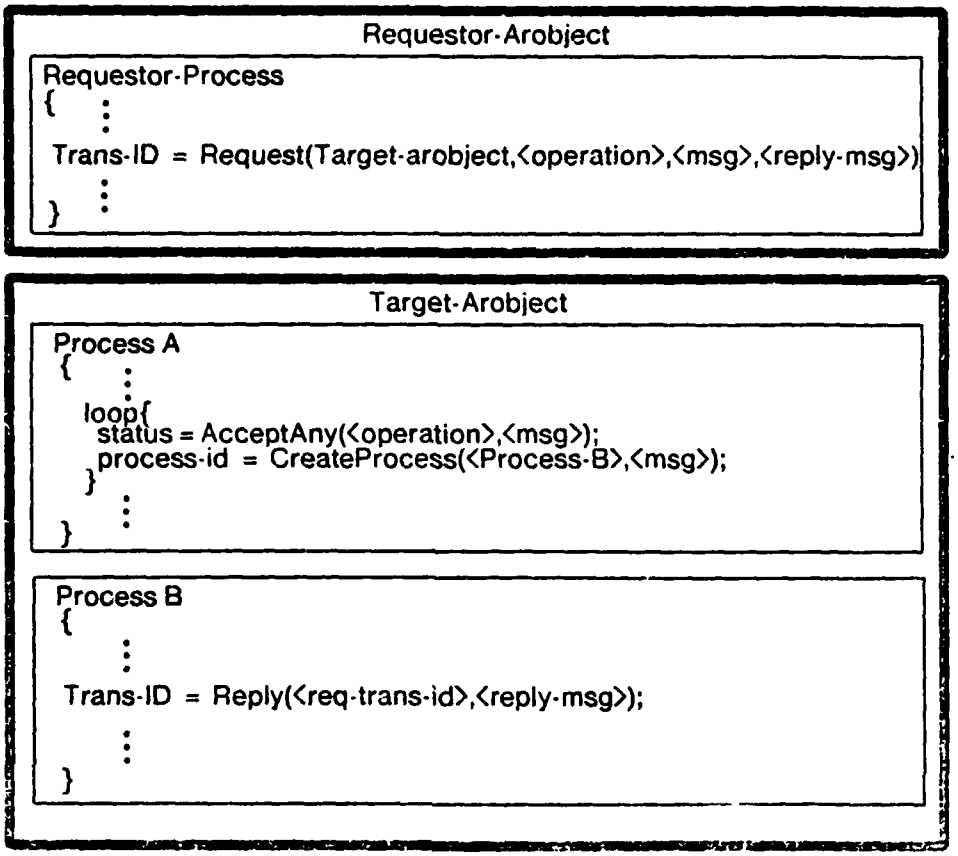


Figure 4-2: Communication Paradigm in ArchOS

4.2.3 System Load and Initialization

Although system generation is beyond the scope of this document, we will assume that a system of application arobjects has been built, and that a directory (possibly partitioned) of the arobjects has also been constructed, and is available to each node. For each distributed program (normally one, but possibly more than one during test phases), one arobject has been identified as a root arobject.

At system startup, once ArchOS has initialized itself and determined its initial state, the directory will be searched for every client root arobject, each of which will then be automatically *Created*, including execution of its INITIAL process. This action will then complete the system load and initialization, with or without operator assistance.

4.2.4 Transactions

ArchOS will use transactions in order to apply the properties that transactions have traditionally displayed in database applications (such as failure atomicity and data permanence) within the operating system. OS-level transactions should facilitate the maintenance of system consistency while simplifying data sharing among multiple processes. In fact, we expect to use transactions extensively within the ArchOS system primitives.

The ArchOS transaction facility does not provide failure atomicity and permanence for all of the data accessed by a transaction. Rather, only the data items that have been explicitly declared to be *atomic* have these properties.

4.2.4.1 Elementary and Compound Transactions

ArchOS will support two types of transactions, elementary transactions and compound transactions, which may be nested in arbitrary combinations. Elementary transactions correspond to traditional nested transactions [Moss 81]. Compound transactions [Sha 84] are supported by ArchOS in order to provide more potential concurrency than elementary transactions and differ from elementary transactions in one important way: when a compound transaction commits, the processing which makes the effects of the transaction permanent and visible (the commit processing) takes place immediately. In this manner, a compound transaction is treated as if it were a top-level transaction, even though it may actually be nested within another transaction.

Compound transactions must be viewed differently than traditional, serializable transactions. In the traditional case, the transaction writer is able to assume that his transaction will be executed as if it were the only transaction in the system. The transaction facility will perform whatever concurrency control is necessary to ensure that the entire transaction will be executed atomically so that no other transaction can view the partial results of this transaction; other transactions can view only the final, committed results. The system thereby ensures that the operations performed on the atomic data objects correspond to some serial ordering of the transactions; this, in turn, guarantees that a set of transactions which individually preserve the consistency of the atomic objects they access will also collectively preserve that consistency.

Compound transactions do not allow the transaction writer to act as though that writer's transaction is the only one active in the system at any given time. Rather, the writer of a compound transaction must realize that, in essense, a compound transaction allows other transactions to view partial results of computations. That is, if a compound transaction is nested within another transaction, when the compound transaction commits, it is implicitly allowing other transactions to view the state of the

atomic data objects that it has manipulated. Since the outer transaction has not yet committed, this state may be considered a partial result of the computation being performed by the outer transaction. (Compound transactions allow partial results to be visible, thus admitting the possibility of side effects, while providing a means of increasing concurrency in applications where such transactions can be employed.)

Due to the fact that a compound transaction can allow other transactions to see partial results of an ongoing computation, compound transactions must be written carefully. In particular, a compound transaction should perform a *consistency preserving* transformation on the set of atomic data objects that it accesses. A consistency preserving transformation is a set of operations that transforms a set of atomic data objects from one consistent state into another consistent state. In this way, no other transaction can ever see a set of atomic data objects in an inconsistent state. For applications in which it is acceptable to allow other transactions to view certain partial results that represent consistent states for a set of atomic data objects, compound transactions may be used, and an increase in overall application and system concurrency can result.

The behavior described above has several important consequences for compound transactions:

- It may be impossible to "undo" (in the sense of Moss's nested transactions) the effects of a committed compound transaction which is nested within another transaction. That is, in the event that a commited compound transaction is later aborted (due to the abortion of a higher level transaction which contains that compound transaction), there is no way that ArchOS can guarantee that the atomic data structures manipulated by that compound transaction can be restored to the same state that they possessed prior to the initiation of the transaction. However, ArchOS does provide a method by which an arobject author can define an operation, called a *compensation operation*, to be associated with each operation defined on a given arobject. It is expected that the arobject writer will write a compensation operation for each arobject operation that could be invoked during a compound transaction. (In some cases, a compensation operation will involve more than simply restoring an object to its original state. For example, after a compound transaction has altered the value of some shared variable, it is possible for other transactions to read, or even change, that value. Later, if compensation must be performed for the committed compound transaction (due to the abortion of a higher-level transaction), it may be necessary to take additional steps to assure that the visibility of the shared variable's value during the interval between the compound transaction's commitment and subsequent abortion has not caused any undesirable side effects. One possible step, for instance, might involve broadcasting a message to all of the potential viewers of the shared variable's value, stating that a transaction abort has caused the value seen most recently to be invalidated.) In the event that one or more arobject operations were invoked during the processing of a committed compound transaction that is subsequently aborted, ArchOS will properly compose the corresponding compensation operations in order to imitate the effects of a traditional "undo" operation. These compensation operations will then transform the states of the atomic data structures manipulated during the compound transaction execution into some member of

a class of states that are *equivalent*, although not necessarily identical, to the pre-transaction states of those data structures.[4]

- Compound transactions can be used to release shared resources before the completion of an entire nested transaction execution, potentially increasing system concurrency.

Transactions are defined by the arobject programmer by means of ArchOS primitives. These primitives appear as programming constructs similar to those used to define *while* and *for* loops. Such a syntactic structure allows a transaction to be placed at any point in any process, while insuring that both the beginning and the end of the transaction occur within a single process.

ArchOS will handle the ordering of compensation operations for aborted compound transactions automatically. This will be done by constructing a sequence of compensation operations corresponding to the operations performed during the execution of a compound transaction. In the event that this compound transaction is committed and subsequently aborted, these compensation operations will be performed in the reverse order of the original execution sequence. As explained above, although ArchOS will initiate the processing of these compensation operations, the operations themselves must be defined by the author of the arobject whose operations were invoked by the compound transaction.

The ArchOS client will often need to access shared data during transaction processing. Such accesses are coordinated by means of a locking mechanism. The client must explicitly request locks on the shared data objects that are needed for the execution of a given transaction. The following section discusses the ArchOS locking facilities in detail.

4.2.4.2 Locks

ArchOS supports two types of locks: *discrete locks* on independent, individual data items, and *tree locks* [Silberschatz 80], which are structures of related discrete locks. In the case of discrete locks, the client obtains (sets) a lock for the desired data item, manipulates the item, and releases the lock.

Tree locks are handled in a somewhat different manner. In this case, there is a tree structure of locks, and there are a few rules that must be followed when accessing the locks. In particular:

- Initially, a client may set a lock located at any point in the tree of locks.
- Subsequently, a client may set any lock whose parent lock is currently held by that client.

---

[4] Strictly speaking, this is not quite true. In fact, the state of the atomic data structures will be equivalent to the state that would have existed had all of the other concurrent committed transactions taken place in the absence of the aborted compound transaction.

- A client cannot acquire a tree lock, release it, and then reacquire it before all of the client's locks in that tree have been released.

- Locks may be released at any time, as long as the above conditions hold.

Tree locks have one important property: if all of the locks involved in a set of computations are contained in a single lock tree and if the above rules are obeyed and if all tree locks are released in a finite amount of time, then deadlocks involving locks in the tree are impossible.

Within transactions, locks are obtained explicitly by the arobject programmer. When a nested elementary transaction commits, its locks are passed to its parent transaction; when a top-level elementary transaction or any compound transaction commits, all of the locks obtained by that transaction are released. (ArchOS will handle this automatically.) When a transaction is initiated from within the scope of a higher-level transaction (see next section for a discussion of transaction scope), it does not automatically inherit the locks which belong to its parent transaction. However, it may attempt to obtain locks held by any of its ancestors by means of the *SetLock* primitive.

In addition to the lock facilities mentioned above, a primitive is provided to release locks. However, the *ReleaseLock* primitive must be used carefully within transactions. Transactions possess desirable properties as a result of the fact that they restrict the freedom with which the locking and unlocking of data items can occur. If a client abuses the locking/unlocking conventions employed by the ArchOS transaction facility by improperly making *ReleaseLock* invocations, then ArchOS may not be able to complete the processing of the client's transaction. Rather, ArchOS will detect the violation of the locking conventions and will terminate the transaction in as orderly a manner as possible. More precisely, ArchOS will abort the transaction upon detection of a lock protocol violation. This may result in an inconsistent or incorrect state for some atomic objects. However, since the transaction can only release locks that it can explicitly name, the integrity of atomic data objects which are manipulated on its behalf can be guaranteed by appropriate use of transactions to encapsulate these other atomic data objects. Such an encapsulation will guarantee that the atomic data objects manipulated by these transactions will be in consistent states and can recover to other consistent states when the lock-violating transaction is aborted. However, since the lock-violating transaction has abused the locking conventions, it may be impossible to properly recover the atomic data objects that it manipulates directly. In that case, ArchOS cannot ensure that these atomic data objects will be in a consistent state. (See the explanation concerning consistency preserving transformations in Section 4.2.4.1.)

Despite the warning given above, the *ReleaseLock* primitive does have "safe" applications. For

instance, it can be used to manipulate tree locks or to release locks that were obtained during non-transaction processing.

4.2.4.3 Transaction Scope, Models, and Lock Passing

This section defines some additional terminology for discussing transactions and introduces the notion of a transaction tree, a structure that allows a simple visualization of the relationships among various transactions. In addition, the rules for lock propagation in the ArchOS transaction facility are specified in more detail.

The preceding discussion concerning transactions dealt with the notion of nested transactions. The transaction tree is a structure that can be used to explain the behavior of a set of transactions. Figure 4-3 contains an example of a transaction tree.

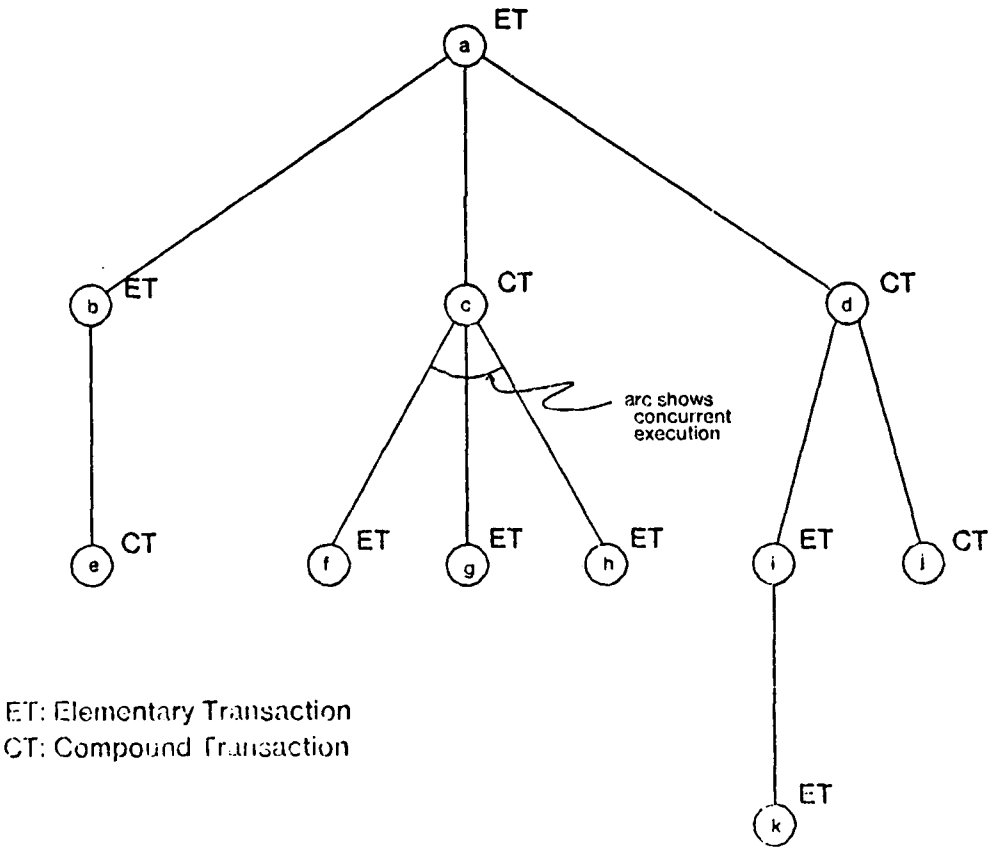


Figure 4-3: Sample Transaction Tree

Each circle in the transaction tree represents a transaction execution. (The transaction nodes have been labeled so that they may be individually referenced later.) Transaction tree nodes that are

children of a given transaction tree node represent transactions that are nested within a parent transaction (that is, either they are contained within the parent transaction's definition or they are executed as a result of invocations performed as part of the parent transaction). Children that are connected to their parent by a single line only are executed in a serial order, with the leftmost child being executed first; children that are connected to their parent by a line through which an arc passes are executed concurrently. (These concurrent executions are typically the result of executing *RequestSingle* or *RequestAll* primitives.)

Each transaction has an associated scope. That scope is delimited by the transaction definition primitive. Any statement that is part of the transaction definition is within the scope of the transaction, as are any statements that are executed as a result of arobject operation invocations or procedure or function calls made by statements within the transaction's scope. With respect to the transaction tree representation, all of the subtree at or beneath the node corresponding to a given transaction lies in the scope of that transaction.

The transaction tree can also be used to explain the ArchOS rules governing lock passing among transactions. As explained in Section 4.2.4.2, the client explicitly obtains the locks that are required by a transaction. When a *nested elementary transaction* commits, all of the locks that it held are passed to the transaction in which it is nested (its parent in the transaction tree); when a nested compound transaction commits, all of its locks are released.

The rules that determine which locks a transaction may obtain are more involved. Two transactions are said to be *unrelated* if: (1) they are not contained in a single transaction tree, or (2) they are in a single transaction tree and are concurrently executing siblings or descendants of concurrently executing siblings, or (3) they are in a single transaction tree in which one is an ancester of the other and either the descendent is a compound transaction or there is a compound transaction on the path connecting the two transaction nodes in the corresponding transaction tree. (Note that according to this definition, a compound child transaction is always unrelated to its parent transaction.) Two unrelated transactions may compete for locks, and they may hold locks with compatible lock modes for a single data object at any given time. However, if they request incompatible lock modes for a single data object, then one of the competitors will obtain a lock and the other will block until it can receive the desired lock, or it will return to the requestor with an appropriate status indication.

Two transactions are said to be *related* if they are contained in a single transaction tree where one is the descendent of the other, the descendent is an elementary transaction, and there are no compound transactions in the path connecting their respective nodes in the transaction tree. The

lock compatibility rules for related transactions are different than those for unrelated transactions. In this case, the descendant transaction can obtain any lock mode for any lock held by a related ancester in the transaction tree, including incompatible lock modes that would not be allowed if the transactions were unrelated. (Of course, the descendant transaction will have to compete with all of the unrelated transactions in the system in order to successfully obtain the requested lock with the desired mode.)

Examples of both related and unrelated transactions can be found in Figure 4-3. For instance, transaction 'k' is related to transactions 'i' and 'd,' but it is unrelated to transaction 'a' (due to point (3) in the above definition of unrelated transactions). Also, while transactions 'f,' 'g,' and 'h' are all related to transaction 'c,' they are all unrelated to one another (due to point (2) in the above definition of unrelated transactions). Finally, there are only two other related transaction pairs in the figure: transaction 'i' is related to transaction 'd' and transaction 'b' is related to transaction 'a.'

4.2.5 Real-Time Facilities

As previously stated, ArchOS is a real-time operating system; for ArchOS, this statement carries a number of important implications. In this work, we define a real-time operating system to be one which manages its system resources to meet user-defined deadlines. Processes will be scheduled using *deadline-driven scheduling* with reference to user-defined policies (See Section 4.2.6). This means that when the system determines that processing resources are sufficient to meet user deadlines at each node, it will meet them, but when resources are insufficient, user policy will guide the operating system in its decisions as to which deadlines should be missed or whether some processes should be relocated.

Examples of user policies to be implemented in the event of insufficient resources include:

- Minimize average lateness
- Minimize maximum lateness
- Minimize number of late processes
- Minimize priorities of late processes

In scheduling terminology, these policies actually define objective functions for the resulting scheduling algorithm(s). Scheduling techniques for some of these objective functions (e.g., minimize maximum lateness) are well known, while for most others optimum algorithms are known to be intractable. ArchOS will use best effort decision making to implement policies such as these and many other similar user policies as closely as possible.

In order to handle real-time constraints, each ArchOS primitive must have its execution time bounded. This bound may be probabilistic, and should be determined with respect to the significant events or actions involved in fulfilling the request (e.g., maximum number of internode messages, maximum cpu time used).

The local scheduling model used by each ArchOS node to manage its real-time load consists of a set of n active processes p_i, i between 1 and n. For each p_i, a stochastic execution time C_i and a deadline T_i are known. C_i is a value estimated by the process' implementer and measured by the system during execution. As a part of the set of user controllable policies described above for handling missed deadlines, a value function V_i will be defined for each processor to determine the value to the system associated with achieving a particular degree of lateness for process p_i. The value functions themselves are either chosen by ArchOS with reference to the user policies, or may be provided directly by the user policies, thus creating an extremely large set of potential overload policies.

The system will continuously monitor its performance with respect to the likelihood of missing currently known deadlines, using a best effort to arrange scheduling to distribute the lateness should missed deadlines occur. This processing is currently a critical portion of the Archons research effort, and the results of this research will be directly applied to this scheduling.

During process scheduling at a single node, an overload condition may be detected in which deadlines may be missed. The scheduling algorithms, under control of the appropriate policies, may determine a (sub)set of processes which should be removed from the local node and moved elsewhere. The decision of where they should be moved will be made by an algorithm to be developed, also under control of the application defined policy.

Primitives are available for specification of periodic process execution, user specified delay, real-time clock management, and lateness doctrine policy specification. With respect to the ArchOS primitives, deadlines are defined by adding the client-defined deadline interval (see *Delay* and *Alarm* primitives, Section 4.3.9) to the request time, which is defined to be: (1) the scheduled periodic process execution time, (2) the expiration time of a *Delay* or *Alarm* primitive currently in progress, (3) the time at which a new process becomes ready for execution following a *CreateProcess* primitive. Processes for which deadlines have not been defined, if any, will be scheduled with the objective of maximum throughput subject to the constraint that deadlines will first be met for processes with deadlines.

4.2.6 Policy Definitions

ArchOS exists to support a distributed (application) program. In order to provide a flexible environment that the program writer can tailor to satisfy specific application-dependent requirements, ArchOS allows the client to define the policies used to manage certain system resources.

In general, the ArchOS client may dynamically specify the policy to be followed with respect to the management of a specific resource. For instance, a process scheduling policy may specify the manner in which processor time is managed. In order to support a wide range of alternative policies, some of which may be defined by the client, ArchOS will include a set of mechanisms that will be combined as appropriate to supply the foundation for the facility that the client has selected.

The Archons researchers are interested in studying the specification of policy by the application programmer, as well as the separation of policy and mechanism within a given resource management facility. In ArchOS, the integration of client-selected policies into the system will be studied in two areas: process scheduling policies and process reconfiguration policies. Process scheduling policies are discussed at some length in this section to provide examples for the ArchOS policy definition facility since these ideas have been actively pursued by Archons researchers; process reconfiguration policies, which govern the dynamic migration of processes from node to node, are not as well developed and are an area of future research.

Consider the definition of a process scheduling policy by the client. There are two different situations under which processes are scheduled: the situation in which there are sufficient processing resources to satisfy all of the client-specified service requests, and the situation in which the demand for processing resources is greater than their supply. At this time, ArchOS policy management is only concerned with the latter case, and this is reflected in the support provided for client-specified scheduling policies.

In the case where the demand for processing resources is greater than their supply, the client has two options: the client may issue a general policy statement which corresponds to a pre-defined scheduling policy, or the client may define a new policy which adheres to the general model that the mechanisms underlying the ArchOS process scheduling facility support. These mechanisms are aimed at optimizing the value of a function, known as the *objective function*. In ArchOS, the objective function will maximize the sum of a *value function* evaluated for each runnable process on a given node. Whenever the client selects a pre-defined scheduling policy, ArchOS will transform this selection into the appropriate value (and hence, objective) function; in situations where the client wants to use a scheduling policy that ArchOS does not "know about," the client must explicitly

specify the value function to be used. (The client accomplishs this by means of specifying the parameters for a value function. Value functions are also discussed in Section 4.2.5.)

The set of pre-defined scheduling policies which may be selected when the demand for processing resources exceeds the supply includes the following examples (See Section 4.2.5 for a discussion of process deadlines and real-time considerations, in general.):

- minimizing the number of deadlines missed;
- minimizing the average lateness;
- minimizing the maximum lateness;
- maximizing the minimum lateness;
- minimizing the weighted tardiness. (Tardiness is defined to be non-negative lateness, and the weighting factor can incorporate process or arobject priorities into the scheduling computation.)

The implementation of client policy definition in ArchOS will include the notions of *policy names* and a corresponding set of *policy modules*. For example, *Schedule* and *Reconfigure* will be recognized by the system as *policy names* and their bodies can be kept as a separate *policy module*. All of the client-defined policy modules are maintained by a *policy set arobject*, which will be instantiated by the application program. Thus, it will also be possible to add, delete, or change the existing policy module for a given policy by invoking a corresponding operation of the policy set arobject during run time.

Each policy module can be built by referring to a set of *policy attributes*. These attributes are used to determine the policy to be carried out by ArchOS or to pass information to the policy module so that it can make a particular policy decision. For instance, the arobject priority and lateness doctrine parameters listed below could be used by ArchOS to translate the client's policy desires into an objective function to handle process scheduling.

Inclusion of a policy set arobject in an application system will result in its INITIAL process being scheduled during system initialization. This arobject may then define a set of policy attributes for this program. Some examples of attributes that might be used for the *Schedule* or *Reconfigure* policies are:

- Arobject priority -- relative scheduling priority of instances of one arobject versus instances of other arobjects in the application program during periods when there are not enough computing resources to satisfy all demands. It should be noted that in the event

of more than one application program running on the ArchOS system, no relative priorities may be determined between them, so performance of either or both may be adversely affected. (As stated in Section 4.2.1.1, ArchOS is designed expressly to support a single program, or set of arobjects. This set of arobjects may well perform computations that would normally be associated with a number of separate "tasks." However, as far as ArchOS is concerned, this collection of tasks comprises a single distributed program since all of the relative priorities among the component arobjects are defined.)

- Lateness doctrine -- rules to be followed in the event that deadlines must be missed. This doctrine indicates the nature of processing which will characterize degraded modes, including such possibilities as maximizing the minimum lateness, minimizing the maximum lateness, minimizing the average lateness, etc. Execution of these doctrines will be done on a best effort basis utilizing the available information, even though incomplete or inaccurate, maximizing its value, and making the decision as near optimal as possible, consistent with the application performance requirements.

4.3 ArchOS Primitives

ArchOS primitives are defined as a set on operations which manipulate various system and kernel arobjects. The primitives can be classified as *system* and *kernel* primitives. The system primitives are provided by system arobjects which exist above the kernel while the kernel primitives are defined within a set of kernel arobjects. In this section, we will specify all of the ArchOS primitives (both kernel and system) which are visible to a client. Other kernel primitives will be described in the ArchOS System Architecture Specification.

4.3.1 Specification of Primitives

We will use the following format to describe the specification of ArchOS primitives. The specification of a primitive consists of two parts. The first part describes the functionality of the primitive, and the second part defines the actual interface for a client process by using a syntactic template shown in below.

It should be noted that since we adopted a C-like syntax in this document, many style of type declataions are adapted from C [Kernighan 78]. For instance, a pointer called "ptr" to a type, say "$TYPE_X$", will be specified as "$TYPE_X$ *ptr". An optional argument, say arg_o, of a primitive will be indicated by [, arg_o] in its argument list.

---

val_0 = Primitive(arg_1, ..., arg_k)

$TYPE_0$ val_0 This indicates that the "val_0" has the type "$TYPE_0$". It also includes the type definition of $TYPE_0$ if necessary.

$TYPE_1$ arg_1 This line specifies the type of $argument_1$.

...

$TYPE_k$ arg_k This line specifies the type of $argument_k$.

The first line in this template shows that an ArchOS primitive, called ***Primitive*** *requires k* arguments, such as arg_1, ..., arg_k, and returns a value.

The additonal part may consists of error or abnormal conditions related to the above primitive invocation and may have the following explanations.

On Error: This part explains the types of errors that can occur and what values will be returned in response to an error. In general, a client can get detailed error information by looking at a specific area called an *error block*. The error block must be declared by using a *SetErrorBlock* primitive (see Section 4.3.11.5).

On Timeout: If a primitive has a timeout argument, then this may explain what happens after a timeout occurs.

This may also contain extra comments such as follows.

Note: This is an example of additional comments on this primitive.

4.3.2 Arobject/Process Management

The arobject and process management provides creation and destruction of arobject instances as well as process instances. The *CreateArobject* and *CreateProcess* primitives create an instance of an arobject and a process respectively. Similary, the *KillArobject* and *KillProcess* primitives kill a specific instance.

The lifetime of an arobject instance can vary, is determined by the lifetime of the last active process instance within that arobject instance. In other words, if an instance of an arobject is killed, all of its internal processes will be halted and removed. A process may terminate by normal exit (i.e., reaching the end of its body) or by an explicit *KillProcess* primitive.

4.3.2.1 Create

A *CreateArobject* primitive creates a new instance of an arobject at an arbitrary node or a specified node. Similarly, a *CreateProcess* primitive creates a new instance of a process in the arobject. The selection of a node is made automatically by ArchOS unless overridden by the *Create* operation. Upon arobject creation, the arobject's INITIAL process is automatically dispatched.

An optional set of parameters can be passed to the INITIAL process when the arobject is instantiated by using an initial message (i.e., "init-msg").

---

arobject-id = **CreateArobject**(arobj-name [, init-msg] [, node-id])
process-id = **CreateProcess**(process-name [, init-msg] [, node-id])

AID arobject-id — The unique identification of the instantiated arobject.

PID process-id — The unique identification of the instantiated process.

AROBJ-NAME arobj-name
The name of arobject to be instantiated.

PROCESS-NAME process-name
The name of process to be instantiated.

MESSAGE *init-msg
A pointer to the initial message which contains initial parameters for the INITIAL process.

NODE-ID node-id — Node identification (optional). An actual node may be designated, or a node selection criterion may be designated (e.g., the current node, any node except the current node, any node, or a specific node).

On Error: If a *CreateArobject* or *CreateProcess* primitive fails, a "NULL-AID" or "NULL-PID" will be returned, respectively. The detailed error code can be found in an error block which is defined by issuing a *SetErrorBlock* primitive. (see Section 4.3.11.5).

4.3.2.2 Kill

The *KillArobject* and *KillProcess* primitives remove a process and arobject instance respectively. An arobject may be killed only by one of its own processes (suicide allowed, no murder). In order to kill another arobject, the target arobject must have an appropriate operation defined within its specification so it can kill itself.

A process can be killed only by a process which exists in the same arobject instance.

---

val = **KillArobject**(aid)
val = **KillProcess**(pid)

BOOLEAN val TRUE if this killing was successful; otherwise FALSE.

AID aid The arobject id of the target arobject to be killed.

PID pid The process id of the target process to be killed.

On Error: If the specified arobject does not exist in the system, or a target process does not exist in the requestor's arobject instance, a "FALSE" value will be returned.

4.3.2.3 SelfID and ParentID

The *SelfAid* primitive returns the requestor's arobject id and the *ParentAid* primitive returns the parent's arobject id of the specified arobject. The *SelfPid* primitive returns the process id (pid) of the requestor and the *ParentPid* primitive returns the parent's pid of the the specified process.

---

aid = **SelfAid**()
paid = **ParentAid**(aid-x)
pid = **SelfPid**()
ppid = **ParentPid**(pid-x)

AID aid, aid-x, paid The arobject id.

PID pid, pid-x, ppid The process id.

On Error: If a non-existing aid or pid is given to *ParentAid/Pid*, a "NULL-AID" or "NULL-PID" will be returned.

4.3.2.4 BindName

Any arobject or process can have a (run time) reference name defined by these binding primitives within a single distributed program. The *BindArobjectName* and *BindProcessName* primitives bind the requested instance of an arobject or process to a reference name. Unless an *Unbind* primitive is executed, the lifetime of a binding is the same as the lifetime of an arobject or process instance.

This binding allows an arobject or a process to have more than one reference name, or a single reference name can be bound multiple arobject or process instances.

To cancel the current binding, a process must use the appropriate unbind primitive.

---

val = **BindArobjectName**(aid, arobj-refname)
val = **BindProcessName**(pid, process-refname)

BOOLEAN val TRUE if this binding was successful.

AID aid The arobject id.

PID pid The process id.

AROBJ-REFNAME arobj-refname
The requested reference name for an arobject given by aid.

PROCESS-REFNAME process-refname
The requested reference name for a process given by pid.

On Error: A "FALSE" value will be returned in the case of an error. For instance, if a client attempts to bind non-existant arobject or process instance to a reference name, a "FALSE" value will be returned.

4.3.2.5 UnbindName

The *UnbindArobjectName* and *UnbindProcessName* primitives release the current binding between the specified instance of an arobject or process and a reference name.

---

val = **UnbindArobjectName**(aid, arobj-refname)
val = **UnbindProcessName**(pid, process-refname)

BOOLEAN val TRUE if this unbinding was successful; otherwise FALSE.

AID aid The arobject id.

PID pid The process id.

AROBJ-REFNAME arobj-refname
The requested reference name for an arobject given by aid.

PROCESS-REFNAME process-refname
The requested reference name for a process given by pid.

On Error: A "FALSE" value will be returned, if a client attempts to release the binding for a non-existing reference name.

4.3.2.6 FindID and FindAllID

A *FindID* primitive returns the unique id (i.e., aid or pid) of the given arobject or process in a specific search domain. A search domain can be specified with respect to all of the internal arobjects, external arobjects, a local node, a remote node, or a reasonable combination of among four. If more than one instance uses the same reference name, the unique id of any one of them will be returned. A *FindAllID* primitive, on the other hand, returns all of the aid's and pid's which correspond to the given reference name.

---

aid = **FindAid**(arobj-refname [, preference])
pid = **FindPid**(process-refname [, preference])
aid-list = **FindAllAid**(arobj-refname [, preference])
pid-list = **FindAllPid**(process-refname [, preference])

AID aid — The arobject id.

PID pid — The process id.

AID-LIST aid-list — The list of corresponding aid's.

PID-LIST pid-list — The list of corresponding pid's.

AROBJ-REFNAME arobj-refname
The reference name of related arobject(s).

PROCESS-REFNAME process-refname
The reference name of related process(es).

PREFERENCE preference
The preference can specify a search domain such as "INTERNAL", "EXTERNAL", "LOCAL", "REMOTE", "INTERNAL-LOCAL", "INTERNAL-REMOTE", "EXTERNAL-LOCAL", "EXTERNAL-REMOTE".

On Error: If the FindAid or FindPid primitive fails, a "NULL-AID" or "NULL-PID" will be returned respectively. If a FindAll primitive fails, a "NULL-AID-LIST" or "NULL-PID-LIST" will be returned.

4.3.3 Communication Management

The communication management provides an inter- and intra-node communication facility among cooperating arobjects. A process can invoke an operation at a specific instance of an arobject by sending a request message or can invoke the same operation at multiple instances of an arobject which have been bound to a single reference name. In the later case, the multiple computations are performed concurrently.

In particular, a process can send any arobject instance a request message to invoke an operation without knowing its actual location. A process can also invoke a private operation which is defined within its local arobject.

There are essentially three types of primitives to provide flexible cooperation among arobjects: *Request*, *Accept*, and *Reply*. In addition to these, the *RequestAll*, *RequestSingle*, and *GetReply* primitives are added in order to interact with multiple instances of arobjects concurrently. All of the communication primitives are executed as transactions so that ArchOS can provide properties such as failure atomicity and permanence. (see Section 4.3.6).

A message consists of a header and a body. The message header will be generated by ArchOS and contains control information. The message body carries all of the parameters and will be set by the client process. Since each arobject has a separate address space, parameters must be sent using call-by-value semantics.

4.3.3.1 Request

The *Request* primitive provides remote procedure call semantics in which the requesting process invokes an operation by sending a message and blocks until the receiving arobject returns a reply message. Identically, if the receiver arobject is the same arobject, a local operation will be invoked. When the reply is generated, it is sent to the requestor which is then unblocked. If the requesting process needs to limit the allowed response time, it may do so by creating a small process to handle the timeout condition.

---

trans-id = Request(arobj-id, opr, msg, reply-msg)

TRANSACTION-ID trans-id
: The transaction id of the transaction on whose behalf the request is being made.

AID arobj-id The unique id of the receiving arobject.

OPE-SELECTOR opr
The name of the operation to be performed.

MESSAGE *msg A pointer to the message which contains the parameters of the operation to be performed. The message to the destination arobject must not contain any pointers (i.e., call-by-value semantics must be used).

REPLY-MSG *reply-msg
A pointer to the reply message.

On Error: The *Request* primitive may fail in the following situations:

- The destination arobject's node is not available.
- The destination arobject does not exist.
- The target operation is not defined.
- The request message does not match the receiver's message type.

If the *Request* primitive fails, a "NULL-TID" will be returned.

4.3.3.2 RequestSingle and RequestAll

The *RequestSingle* and *RequestAll* primitives can send a request message and proceed without waiting for a reply message. The *RequestSingle* primitive provides nonblocking one-to-one communication and, the *RequestAll* primitive supports one-to-many communication. The requesting process may thus invoke an operation on more than one instance of an arobject or process with one request. To receive all of the replies, the *GotReply* primitive may be repeated until a reply with a null body is received.

---

trans-id = RequestSingle(arobj-id, opr, msg)
trans-id = RequestAll(arobj-refname, opr, msg)

TRANSACTION-ID trans-id
The transaction ID of the transaction on whose behalf the request is being made.

AID arobj-id The ID of the receiving arobject.

AROBJ-REFNAME arobj-refname
The reference name of the receiving arobject(s).

OPE-SELECTOR opr
The name of the operation to be performed.

MESSAGE *msg A pointer to the message which contains the parameters of the operation to be performed. The message to the destination arobject must not contain any pointers (i.e., call by value semantics must be used).

On Error: The *RequestSingle* and *RequestAll* primitives fail if similar to those situations mentioned in the previous section happen. If the *RequestSingle* or *RequestAll* primitive fails, a "NULL-TID" will be returned.

4.3.3.3 GetReply

The *GetReply* primitive receives a reply message which has the specific transaction id generated by the preceding *RequestAll* primitive. If the specific reply message is not available, then the caller will be blocked until the message becomes available.

---

aid = **GetReply**(req-trans-id, reply-msg)

AID aid The arobject id of the replying arobject.

TRANSACTION-ID req-trans-id
The transaction id of the corresponding *RequestSingle* or *RequestAll* primitive.

REPLY-MSG *reply-msg
A pointer to the reply message.

On Error: The GetReply primitive fails, if the specified transaction does not exist; a "NULL-AID" will then be returned.

4.3.3.4 Accept and AcceptAny

The process responsible for an arobject operation receives a message using the *Accept* primitive. Using the colection criteria specified, the operating system selects an eligible message from the arobject's input queue and returns it. The process operates on the message, responding with a reply when processing has been completed. If no suitable message is in the request message queue, then the caller will block until such a message becomes available.

The *AcceptAny* primitive can receive a message from any arobject instance with any operation (i.e.,

"ANYOPR") or a specified operation request. The primitive can return the requestor's transaction id, specified operator, and requestor's aid. The *Accept* primitive can receive a message from a specific requestor arobject and returns the requestor's transaction id and the requested operator.

(req-trans-id, req-opr, requestor) = **AcceptAny**(opr, msg)

(req-trans-id, req-opr) = **Accept**(requestor, opr, msg)

AID requestor The aid of the requesting arobject.

OPE-SELECTOR opr, req-opr
The name of operation to be performed. The "opr" parameter can be a specific operation name or "ANYOPR".

TRANSACTION-ID req-trans-id
The transaction id of the transaction on whose behalf the request is made.

MESSAGE *msg A pointer to the message buffer.

On Error: The *AcceptAny* primitive fails if the specified operation does not exist. Similarly, the *Accept* primitive fails if the requestor or the operation does not exist. If the *AcceptAny* or *Accept* primitive fails, a "NULL-TID", "NULL-OPR", and "NULL-AID" will be returned.

4.3.3.5 CheckMessageQ

The CheckMessageQ primitive examines the current status of an incoming message queue without blocking the caller process. The primitive must specify a message queue type, either "request-queue" or "reply-queue". The request-queue queues all of the non-accepted request messages and is allocated for each arobject instance. The reply-queue maintains all of the non-read reply messages and is assigned to every process instance.

A message can be selected based on the sender's arobject id, operation name, and/or transaction id. If more than one argument is given, only messages which satisfy all of the conditions will be returned. If no corresponding message exists in a specified message queue, a "NULL-POINTER" will be returned.

---

ptr-mds = CheckMessageQ(qtype, requester, opr, req-trans-id)

MSG DESCRIPTORS *ptr-mds

Pointer to a list of the message descriptors selected by the specified selection criteria.

MSG-Q qtype This indicates either "request-" or "reply-" message queue.

AID requestor The aid of the requesting arobject.

OPE-SELECTOR opr
The operation to be performed. The "opr" parameter can be a specific operation name or "ANYOPR".

TRANSACTION-ID req-trans-id
The transaction id of the corresponding *RequestSingle* or *RequestAll* primitive.

On Error: The *CheckMessageQ* primitive fails if the specified message queue, operation, or the transaction id does not exist. If the *CheckMessageQ* primitive fails, a "NULL-POINTER" will be returned.

4.3.3.6 Reply

After processing an *Accept* primitive, a process must use a *Reply* primitive to send the completion message to the requesting process. At the requestor's site, the completion message is received by the second half of the synchrouns *Request* primitive or the *GetReply* primitive. It should be noted that the reply need not necessarily be sent from the same process which accepted the operation. In other words, the requestor's transaction id is used to determin a proper reply message.

---

trans-id = **Reply**(req-trans-id, reply-msg)

TRANSACTION-ID trans-id
The transaction id of this *Reply* primitive (not the requestor's transaction id)

TRANSACTION-ID req-trans-id
The transaction id of the request that has been serviced.

MESSAGE *reply-msg
A pointer to the reply message.

On Error: The *Reply* primitive fails if the specified transaction has already been aborted. If the *Reply* primitive fails, a "NULL-TID" will be returned.

4.3.4 Private Object Management

A process can dynamically create a private object, which is an instance of a private abstract data type defined in its arobject, at any node. If a private abstract data type has instances of atomic or permanent data objects, the actual data objects will be allocated in non-volatile memory.

4.3.4.1 Allocate/Free Object

An *AllocateObject* primitive allocates an instance of a private abstract data type at any node and a *FreeObject* primitive deallocates the specified instance.

---

object-ptr = **AllocateObject**(object-type, parameters [, node-id])

val = **FreeObject**(object-ptr)

OBJECT-PTR object-ptr
A pointer to the allocated private data object.

OBJECT-TYPE object-type
The object-type indicates the name of a private abstract data type.

BOOLEAN val TRUE if the object was released successful; otherwise FALSE.

NODE-ID node-id Node identification. An actual node may be designated, or a node selection criterion may be designated (e.g., the current node, any node except the current node, any node, or a specific node).

On Error: If the object-type is not defined an *AllocateObject* primitive fails and returns a "NULL-POINTER". If the object-ptr is not pointing to a proper permanent object, then the *FreeObject* primitive fails and returns "FALSE".

4.3.4.2 FlushPermanent

A *FlushPermanent* primitive blocks the caller until the specified data object is saved in non-volatile storage.

FlushPermanent(object-ptr, size)

OBJECT-PTR object-ptr
A pointer to the permanent data object.

INT size The number of bytes which must be flushed into permanent storage.

On Error: The *FlushPermanent* primitive fails if the specified object does not exist.

4.3.5 Synchronization

ArchOS provides two levels of synchronization facilities. One, the critical region, is for controlling the concurrent access to a single shared object within an arobject, while the other, the locak, is for controlling accesses from concurrent transactions.

The critical region scheme should be used when the shared object does not need to provide failure atomicity. That is, a client may be able to access an inconsistent state of the object. On the other hand, if these objects are atomic objects and are accessed from transactions, then a client must be allowed to see only consistent objects. In the critical region scheme, mutual exclusion is achieved by implicit locking by using an event variable, while the transactions require an explicit lock on the atomic object. The *CreateLock* and *DeleteLock* primitives are provided to create and remove a lock for the explicit locking scheme. Actual locks can be set by using a *SetLock* or *TestandSetLock* primitive and can be released by using a *ReleaseLock* primitive.

4.3.5.1 Region

The *Region* construct provides a simple mutual exclusion mechanism for controlling concurrent accesses to shared objects. A timeout value must be specified in order to bound the total execution time in the critical region including any time spent waiting to enter the region. If a timeout occurs, a client process is forced to exit from the critical region and an error status is returned.

Region(ev, timeout){ . . . }

EVENT-VAR ev — The event variable consists of a waiting queue of client processes and an event counter.

TIMEOUT timeout — The timeout value should indicate the maximum execution time for this critical region including the waiting time.

On Timeout: A client should be able to check whether the critical region was exited due to a timeout by checking error information in its error block (see Section 4.3.11.5).

4.3.5.2 CreateLock and DeleteLock

The *CreateLock* primitive creates either a *tree-type* or *discrete-type* lock and returns a unique *lock id*. Since a lock id is not bound to any data object explicitly, a client must be responsible for utilizisng the lock in accordance with the locking protocol. The *DeleteLock* primitive removes the specified lock from the system.

---

newlock-id = **CreateLock**([parent-lockid])
val = **DeleteLock**(lockid)

LOCK-ID newlock-id
A new lock id will be returned.

BOOLEAN val TRUE if the lockid is removed successfully; otherwise FALSE.

LOCK-ID parent-lockid
If the created lock must be a tree-type lock, then its parent-lockid must be specified. If the parent-lockid is a "NULL-LOCK-ID", then the new lock will be the root of a new lock tree. If a parent-lockid is not given, then the new lock will be a discrete-type lock.

LOCK-ID lockid The lockid to be deleted. If the lockid is a tree-type lock, then the entire subtree of which this lock is the root will be deleted.

On Error: If a parent lockid does not exist, then the *CreateLock* primitive fails and returns a "NULL-LOCK-ID". A *DeleteLock* primitive fails if the specified lockid does not exist, and returns "FALSE".

4.3.5.3 SetLock, TestLock, and ReleaseLock

The *SetLock* primitive sets a "tree-type" or "discrete-type" lock on arbitrary objects by specifying a lock key and its mode. If a requested lock is being held, the caller will block until it is released. The *TestandSetLock* primitive also tries to set a lock, however, it will return a "FALSE" if the lock is being held. If the request lock is a tree-lock type, then the *SetLock* and *TestandSetLock* primitives may also fail due to the violation of the tree-lock convention (See Section 4.2.4.2).

The *TestLock* primitive checks the availability of a specified lock with a lock mode. In the case of a tree lock, it also checks whether the locking would be legal in the corresponding lock tree. The *ReleaseLock* primitive can release the lock on an object which was gained by the *SetLock* or *TestandSetLock* primitive explicitly.

---

sval = SetLock(lock-type, lockid, lock-mode)
sval = TestandSetLock(lock-type, lockid, lock-mode)
tval = TestLock(lock-type, lockid, lock-mode)
rval = ReleaseLock(lock-type, lockid, lock-mode)

INT sval — 1 if the specified lock is set; 0 if the lock is not set. A negative value will be returned if an error occured.

INT tval — 1 if the specified lock is being held; 0 if the lock is not being held. A negative value will be returned if an error occured.

INT rval — 1 if the specified lock is released; 0 if the lock is not released. A negative value will be returned if an error occured.

LOCK-TYPE lock-type
The lock type can be either "TREE" or "DISCRETE".

LOCK-ID lockid — The lockid indicates the unique id of a lock.

LOCK-MODE lock-mode
The lock mode can be "READ", "WRITE", etc.

On Error: An error may occur if a non-existent lock id or lock mode is used for the above primitives. A detailed error condition, such as a non-existent lock or lock mode, is available by looking at the caller's error block (see Section 4.3.11.5).

4.3.6 Transaction/Recovery Management

ArchOS can allow a client process to create a compound transaction or an elementary transaction which can be nested in any combination. By using nested elementary transactions, a client can use a traditional "nested transactions" mechanism. In addition to this, the compound transaction provides a mechanism which can commit the transactions at the end of the current scope without delaying the commit point to end of the top-level transaction. Since a completed nested compound transaction cannot be undone, ArchOS provides a mechanism to perform corresponding compensate actions automatically (see Section 4.2.4).

It should be noted that ArchOS cannot generate such a sequence of compensate actions automatically. However, ArchOS provides a mechanism to execute the defined compensate actions in the proper sequence to make the status of each affected atomic object into a member of an *equivalence class* of its correct "pre-execution" state. (See Section 4.2.4.1)

4.3.6.1 Compound Transaction

A compound transaction construct creates a new transaction scope in a client process. Within this scope, a client can access atomic objects as if these computational steps were executed alone.

When a compound transaction starts, no locks will be inherited from its parent transaction if one exists. That is, all of its locks must be obtained within this scope by means of the *SetLock* primitives. (See Section 4.3.5.3). However, at the end of the compound transaction scope, ArchOS releases all of the locks for this transaction automatically. It is also possible to release locks before the end of the transaction scope by using a *ReleaseLock* primitive explicitly.

If a compound transaction must abort, an *AbortTransaction* primitive (see Section 4.3.6.4) performs the necessary compensate actions, breaks the current transaction scope, and passes control to the end of the transaction scope.

CT(timeout){ . . . <transaction steps> . . . }

TIME timeout The timeout value indicates the maximum lifetime of this compound transaction.

<transaction steps> Atomic objects can only be accessed and altered, within these transaction steps.

On Timeout: The current transaction and all of its child transactions will be aborted. That is, ArchOS will execute all of the necessary compensate actions and undo. After completion of these actions, the status of atomic objects should be consistent and be one of the member of the *equivalence class* of its pre-(transaction) execution state.

4.3.6.2 Elementary Transaction

An elementary transaction construct also creates a new transaction scope in a client process. Within this scope, a client can access atomic objects as if these computational steps were executed alone.

When an elementary transaction starts, it can obtain its ancestor transaction's locks if there is an ancestor. That is, this elementary transaction may access atomic objects which were manipulated by the higher level transactions. If the ancestor has modified atomic objects, these modifications will be visible to this transaction. At the end of an elementary transaction scope, ArchOS will propagate all of its locks to the parent transaction if one exists. If the elementary transaction is the topmost

transaction, then all of its locks will be released at this point and all of its atomic objects will be committed.

If an elementary transaction must abort, an *AbortTransaction* primitive (see Section 4.3.6.4) performs the necessary "undo" actions, breaks the current transaction scope, and passes the current control to the end of the transaction scope.

ET(timeout){ . . . <transaction steps> . . . }

TIME timeout — The timeout value indicates the maximum lifetime of this elementary transaction.

<transaction steps> — Atomic objects can only be accessed and altered, within these transaction steps.

On Timeout: The current transaction and all of its child transactions will be aborted. That is, ArchOS will execute all of the necessary undo and compensate actions automatically. After completion of these actions, the status of all affected atomic objects should be consistent and be "identical" to the correct initial (pre-execution) states.

4.3.6.3 SelfTid and ParentTid

The *SelfTid* primitive returns the id of the current transaction and the *ParentTid* primitive returns the parent transaction id of the given transaction id.

```
mytid = SelfTid()
ptid = ParentTid(tid)
```

TID mytid — The id of the current transaction.

TID tid — The id of the specific transaction.

TID ptid — The parent's tid of the given transaction "tid".

On Error: If the *ParentTid* primitive fails, a "NULLTID" will be returned.

4.3.6.4 AbortTransaction

The *AbortTransaction* primitive aborts the specified transaction and all of its child transactions within the same transaction tree (See Figure 4-3 in Section 4.2.4.3). If the transaction that invokes the *AbortTransaction* primitive does not belong to same the transaction tree as the transaction which is to be aborted, a client cannot abort that transaction. This primitive executes all of the necessary "undo" or "compensate" actions, based on the transaction type, and breaks the current transaction scope. After completion of these actions, the status of all affected atomic objects will be consistent and returned to either "identical" to or "a member of the equivalence class" of their initial (pre-execution) states.

The *AbortIncompleteTransaction* primitive also aborts all of the outstanding incomplete transactions which had been initiated by an outstanding *RequestSingle* or *RequestAll* primitive. In other words, all of the nested transactions which belong to the specified request transaction but have not yet completed (committed) will be aborted.

---

val = AbortTransaction(tid)
val = AbortIncompleteTransaction(req-tid)

BOOLEAN val — TRUE if the transaction was aborted successfully; otherwise FALSE.

TID tid — The id of the transaction.

TID req-tid — The transaction id of the *RequestSingle* or *RequestAll* primitive.

On Error: If an *Abort* primitive is called from a transcation which is not a parent of, or identical to, the designated transaction, the primitive will fail and return FALSE.

4.3.6.5 TransactionType

The *TransactionType* primitive returns the type of the given transaction (a compound or elementary) and also indicates the transaction level.

---

trantype = TransactionType(tid)

TRANTYPE trantype — The type of the given transaction, such as "CT", "ET", "Nested CT", or "Nested ET".

TID tid — The id of the transaction.

On Error: If there is no specified transaction in its transaction tree, the TransactionType primitive fails and returns "NULL-TRAN-TYPE".

4.3.6.6 IsCommitted

The *IsCommitted* primitive checks whether the given transaction is already committed or not.

---

val = **IsCommitted**(tid)

BOOLEAN val	TRUE if the specified transaction was committed; otherwise FALSE.
TID tid	The id of the transaction.

On Error: If there is no specified transaction in its transaction tree, the *IsCommitted* primitive fails and returns "FALSE".

4.3.6.7 IsAborted

The *IsAborted* primitive checks whether the given transaction is already aborted or not.

---

val = **IsAborted**(tid)

BOOLEAN val	TRUE if the specified transaction was aborted; otherwise FALSE.
TID tid	The id of the transaction.

On Error: If there is no specified transaction in its transaction tree, the IsAborted primitive fails and returns "FALSE".

4.3.7 File Management

Viewing a file as a set of long term persistent data, it is clear that an arobject can also fulfill this role. To create a file, an instance of the appropriate arobject can be created, and the filename can be bound to it. To erase the file, the *Kill* primitive will serve. Reading from and writing to the file can be performed using the appropriate operations of the arobject itself. Similarly, control functions (e.g. backspace, random placement, search) become operations of the arobject. The data to be stored in

the file is merely contained in one of the arobject's *private data objects*. If this private data object is declared to be atomic, then the file will be treated as an *atomic* file and all of the file accesses must be performed from within a transaction scope.

4.3.7.1 File Access Interface

In order to avoid the low level (bare) access to a file arobject (i.e., an explicit invocation of an operation on a file arobject), ArchOS provides the following set of primitives which support conventional file access and control functions.

The *OpenFile* primitive opens a specified file with the given access mode, and the *CloseFile* file primitive closes the file. The *ReadFile* and *WriteFile* primitives provide a simple "byte-stream" oriented read and write access to a file, respectively. The *CreateFile* primitive creates a file of a given file type and the *DeleteFile* primitive deletes a file. The *SeekFile* primitive moves the current reading or writing position to a specified by bytelocation in the file.

```
fd  = OpenFile(filename, mode)
val = CloseFile(fd)
nr  = ReadFile(fd, buf, nbytes)
nw  = WriteFile(fd, buf, nbytes)
val = CreateFile(filename, filetype)
val = DeleteFile(filename)
pos = SeekFile(fd, offset, origin)
```

FILEDESCRIPTOR *fd
: A pointer to the file descriptor.

BOOLEAN val
: TRUE if the specified operation is done successfully; otherwise FALSE.

INT nr
: The actual number of bytes which were read.

INT nw
: The actual number of bytes which were written.

FILENAME filename
: The name of the file.

ACCESSMODE mode
: The mode for accessing the file. (e.g., "READ", "WRITE", "APPEND", "READLOCK", "WRITELOCK", etc).

FILE-TYPE filetype
: The type of the specified file such as "ATOMIC", "PERMANENT" or "NORMAL".

BUFFER *buf
: The buffer address.

INT nbytes The number of bytes to be read or written during a Read or Write operation, respectively.

LONGINT pos The current pointer's position in the file in bytes. If the seek action is done successfully, the value of pos must be a positive integer; otherwise -1.

FILEOFFSET offset The offset value from the given origin point, in bytes.

FILEORIGIN origin The origin indicates the origin of the seek operation. (At the beginning, the current position, or the end of the file).

On Error: If the file does not exist or the file mode is not supported, then the *OpenFile* primitive fails and returns "NULL-FILE-DESCRIPTOR". If the file descriptor is not an opened descriptor, then a close or read/write action fails and "-1" will be returned and a detailed error condition will be also set in the caller's error block. If a create/delete action fails, then "FALSE" will be returned and a detailed error condition will be also set in the caller's error block. If the specified file does not exist or the value of offset or origin is not in the proper range, "-1" will be returned and a detailed error condition will be also set in the caller's error block.

4.3.8 I/O Device Management

A normal I/O device access protocol should be similar to the file access protocol described in Section 4.3.7. To read, write or send a special command, the target device must be opened. Once all actions are done, it must to be closed. All device dependent commands can be sent to devices by using the *SetIOControl* primitive.

At the lowest level, I/O control and data transfer will be handled according to the hardware interface definition (e.g., memory read and write to memory mapped devices). The *IOWait* primitive can be used to wait for an interrupt from a particular device; at most one process may wait for a particular device interrupt at one time.

4.3.8.1 Basic I/O Device Access Interface

Typically, access to a device is performed by the following primitives. If a device is not readable or writeable, then a special *SetIOControl* primitive (see Section 4.3.8.3) must be used.

```
dd  = OpenDevice(device-name, mode)
val = CloseDevice(dd)
nr  = ReadDevice(dd, buf, nbytes)
nw  = WriteDevice(dd, buf, nbytes)
```

DEVDESCRIPTOR *dd
A pointer to the device descriptor.

BOOLEAN val TRUE if the specified operation is done successfully; otherwise FALSE.

INT nr The actual number of bytes which were read.

INT nw The actual number of bytes which were written.

DEVNAME device-name
The device name.

ACCESSMODE mode
The access mode of the specified device such as "READ", "WRITE", etc.

BUFFER *buf The buffer address.

INT nbytes The number of bytes to be read or written during a *ReadDevice* or *WriteDevice* operation, respectively.

On Error: If the device does not exist or the device mode is not supported, the *OpenDevice* primitive fails and returns "NULL-DEV-DESCRIPTOR". If the device descriptor is not an opened descriptor, then a close or read/write action fails and "-1" will be returned, a detailed error condition will be set in the caller's error block.

4.3.8.2 IOWait

The *IOWait* primitive blocks the requestor process until the specified device completes an I/O action.

---

event-cnt = IOWait(dd, timeout)

INT event-cnt An event counter which indicates the number of basic I/O actions performed. If a negative number is returned, it indicates an error state.

DEV-DESCRIPTOR dd
The device descriptor.

TIME timeout The timeout value indicates the maximum execution time of this I/O function.

On Error: If the device descriptor is not an opened descriptor, a wait action fails, "-1" will be returned, a detailed error condition will be set in the caller's error block.

On Timeout: If an IOWait primitive cannot complete within the specified timeout value, a negative value will be returned.

4.3.8.3 SetIOControl

The *SetIOControl* primitive sends device control information to the specified device. It also receives status information from the device.

---

val = **SetIOControl**(dev-descriptor, io-command, dev-buf, timeout)

BOOLEAN val TRUE if this SetIOControl succeeded; otherwise FALSE.

DEV-DESCRIPTOR *dev-descriptor
A pointer to the device descriptor.

IO-COMMAND io-command
The io-command indicates a device-specific control command.

DEV-BUF *dev-buf The dev-buf indicates a pointer to a buffer which will be filled with device status.

TIME timeout The timeout value indicates the maximum execution time of this I/O function.

On Error: If the device descriptor is not an opened descriptor, then a control action fails and "FALSE" will be returned and a detailed error condition will be also set in the caller's error block.

On Timeout: If an *SetIOControl* primitive cannot complete within the specified timeout value, a negative value will be returned.

4.3.9 Time Management

In ArchOS, the time management not only provides a basic access to the system real time clock which is maintained in non-volatile storage, but also supports primitives which provide the basic functions of the time-driven (process) scheduling.

The *GetRealTime* primitive obtains the system's current real time clock value, while the *GetTimeDate* primitive fetches the current absolute time and date. The *Delay* primitive delays the caller's execution for the specified time period. The *Alarm* primitive also postpones the caller until the specified time of day and date. The *Delay* and *Alarm* primitives provide aperiodic time dependent processing control, while periodic repetitive processing will normally be controlled using the Policy primitives (see Section 4.3.10).

4.3.9.1 GetRealTime

The *GetRealTime* primitive returns the current real time clock value in microseconds.

---

rtc = **GetRealTime()**

REALTIME rtc — Value of current real-time clock in microseconds. This value may be used to compute time values for use in delay or alarm primitives.

4.3.9.2 GetTimeDate

The *GetTimeDate* primitive returns the current absolute time and date. The time value accuracy will be limited by delays in the calibration entry performed by the application, and cannot be expected to return exactly the same value simultaneously at every node. ArchOS will maintain this information on a best effort basis.

(time, date) = **GetTimeDate()**

TIME time — Value of current time of day in microseconds from midnight.

DATE date — Julian date of this day.

On Error: If the *SetTimeDate* primitive operation has not been executed since the system was initialized, the date value returned is zero.

4.3.9.3 Delay

The *Delay* primitive delays a specified length of time (in microseconds), then returns the current date and time when the delay has been completed and execution has resumed. This primitive can also be used to specify a deadline, by which time a deadline milestone must be reached, as well as an estimate of the amount of processing time that will be required to reach the deadline milestone. (ArchOS may also make estimates about this processing time based on observations of earlier executions.) Finally, the client may use the *Delay* primitive to mark the arrival of the process at the deadline milestone referred to as the current deadline (while optionally defining the next deadline milestone).

(time, date) = **Delay**(delaytime, deadline, util, dlflag)

REALTIME delaytime
: Time to be delayed starting at the present time, in microseconds. No delay if this value is not positive; in this case it replies immediately.

REALTIME deadline
: Elapsed time in microseconds from delaytime by which the deadline milestone must be reached. If this value is not positive, the deadline remains unchanged.

REALTIME util
: Estimated execution time of this process in microseconds which will be required before the deadline is reached. If this value is 0 or less, the current processing time estimate remains unchanged. The use of this value by ArchOS is defined by the policy mechanism (see Section 4.2.6).

TIME time
: Current time of day in microseconds since midnight.

DATE date
: Julian date of this day.

BOOLEAN dlflag
: Client sets to "TRUE" if this call is intended to mark the arrival of this process at the milestone referred to as the current deadline.

On Error: If the *SetTimeDate* primitive operation has not been executed since the system was initialized, the date value returned is zero.

4.3.9.4 Alarm

The *Alarm* primitive waits until the specified time of day and date, then replies with the current time and date. If the specified time has already passed, it replies immediately. (The accuracy of the day and date is as described in the *GetTimeDate* primitive above.) Like the *Delay* primitive, the *Alarm* primitive can also be used to specify a deadline, by which time a deadline milestone must be reached, as well as an estimate of the amount of processing time required to reach the deadline milestone. (ArchOS may also make estimates about this processing time based on observations of earlier executions.) Finally, the client may use the *Alarm* primitive to mark the arrival of the process at the deadline milestone referred to as the current deadline (while optionally defining the next deadline milestone).

(time, date) = **Alarm**(alarmtime, deadline, util, dlflag)

REALTIME alarmtime
: Time and Date at which processing of this process is requested to resume.

REALTIME deadline
: Elapsed time in microseconds from delaytime by which the deadline milestone must be reached. If this value is not positive, the deadline remains unchanged.

REALTIME util
: Estimated execution time of this process in microseconds which will be required before the deadline is reached. If this value is 0, the current processing time estimate remains unchanged. The use of this value by ArchOS is defined by the policy mechanism (see Section 4.2.6).

TIME time
: Current time of day in microseconds since midnight.

DATE date
: Julian date of this day.

BOOLEAN dlflag
: Client sets to "TRUE" if this call is intended to mark the arrival of this process at the milestone referred to as the current deadline.

On Error: If the *SetTimeDate* primitive operation has not been executed since the system was initialized, the date value returned is zero.

4.3.9.5 SetTimeDate

The SetTimeDate primitive sets the current absolute time and date. The date will be written into the ArchOS data base and continually adjusted by ArchOS. ArchOS will maintain this information on a best effort basis. This primitive should be used only once, when the application is initiated.

---

val = **SetTimeDate**(time, date)

BOOLEAN val
: TRUE if time and date were set; otherwise FALSE.

TIME time
: Value of current time of day in microseconds from midnight.

DATE date
: Julian date of this day.

On Error: If the time and date were already set by this primitive, they remain unchanged, and val is set to "FALSE".

4.3.10 Policy Management

In ArchOS, a policy management is carried out by a *policy set arobject* and a set of *policy modules.* The policy set arobject must exist in a distributed program and maintain a directory of the policy modules. The actual policy will be implemented as a separate policy module in ArchOS. A set of *policy attributes* are also maintained by the policy set arobject and ArchOS and will be to referred by the policy modules.

A *SetPolicy* primitive specifies a new policy module to carry out a designated policy in the applications's policy set arobject. If no *SetPolicy* primitive is invoked, ArchOS provides a default policy module for the application program. A *SetAttribute* primitive sets an attribute into a new value.

---

val = **SetPolicy**(policy-name, policy-module)

val = **SetAttribute**(attr-name, attr-value)

BOOLEAN val TRUE if the specified policy was set properly; otherwise FALSE.

POLICY-NAME policy-name
The name of the policy to be set. (Well-known name to ArchOS, such as *SCHEDULE*.)

POLICY-MODULE policy-module
The name of the policy module which contains the policy.

ATTRIBUTE-NAME attr-name
The name of attribute to be set.

ATTRIBUTE-VALUE attr-value.
The actual value for the attribute.

On Error: If there is no policy name or policy body, the SetPolicy primitive fails and a "FALSE" will be returned. The SetAttribute primitive fails if a specified attribute is not defined or inappropriate values is assigned.

4.3.11 System Monitoring and Debugging Support

ArchOS provides system monitoring and debugging primitives in order to control the complexity of application development in a distributed environment. A system monitoring facility can provide for tracking the behavior of arbitrary arobjects or processes in the system. The communication activity among arobjects can be also monitored by intercepting the selective message.

4.3.11.1 Freeze and Unfreeze

A *FreezeAllApplications* primitive stops the entire activities of caller's application, and a *FreezeNode* primitive halts all of the cleint's activities in a specific node. To resume client's application, *UnfreezeAllApplications* or *UnfreezeNode* will be used.

A *FreezeArobject* primitive stops the execution of an arobject (i.e., all of its processes), and a *FreeseProcess* primitives halts a specific process for inspection. An *UnfreezeArobject* and *UnfreezeProcess* primitive resumes a suspended arobject and process respectively. While a process is in a *frozen* state, many of the factors used for making scheduling decisions can be selectively ignored. For instance, a timeout value will be ignored by specifying a proper flag in the *Freeze* primitive.

---

val = **FreezeAllApplications()**
val = **UnFreezeAllApplications()**
val = **FreezeNode**(node-id)
val = **UnfreezeNode**(node-id)
val = **FreezeArobject**(arobj-id [, options])
val = **UnfreezeArobject**(arobj-id [, options])
val = **FreezeProcess**(pid [, options])
val = **UnfreezeProcess**(pid [, options])

BOOLEAN val — TRUE if the primitive was executed properly; otherwise FALSE.

NODE-ID node-id — The node id indicates the actual node which wil be stopped.

AID arobj-id — The unique arobject id of an arobject instance.

PID pid — The process id of the target process.

FREEZE-OPT options
The options indicate various selectable flags such as a timeout freeze/unfreeze flag.

On Error: An error condition, such as non-existent node id, arobject id or pid, will be noted, and detailed information regarding the error status will be available by looking at the caller's error block. If the target node, arobject, or process is already unfrozen, then the *Unfreeze* action will be ignored and FALSE will be returned along with the appropriate error information is in the error block.

4.3.11.2 Fetch and Store Arobject and Process' Status

A *Fetch* primitive inspects the status of a running or frozen arobject or process in terms of a set of frozen values of private data objects. The specific state of the arobject or process will be selected by a data object id. The state includes not only the status of private variables, but also includes process control information.

---

fval = **FetchArobjectStatus**(arobj-id, dataobj-id, buffer, size)
sval = **StoreArobjectStatus**(arobj-id, dataobj-id, buffer, size)
fval = **FetchProcessStatus**(pid, dataobj-id, buffer, size)
sval = **StoreProcessStatus**(pid, dataobj-id, buffer, size)

INT fval — The actual number of bytes which were fetched.

INT sval — The actual number of bytes which were stored.

AID arobj-id — The unique id of the arobject instance.

PID pid — The process id of the target process.

DATAOBJ-ID dataobj-id — *The dataobj-id indicates the private object or system control status of the* target arobject/process.

BUFFER *buffer — A pointer to the buffer area for storing the returned data object value.

INT size — The size indicates the buffer size in bytes.

On Error: An error condition, such as non-existent arobject id or pid, will be noted, and detailed information regarding the error status will be available by looking at the caller's error block. If the fetched data object is larger than the specified buffer size, then the content will be truncated. For the storing operations, the data object size must be equal, otherwise the value will not be replaced.

4.3.11.3 Kill Arbitrary Arobject/Process

A *GlobalKill* primitive can destroy an arbitrary arobject or process in the system.

---

nproc = **GlobalKillArobject**(arobj-id [, options])
val = **GlobalKillProcess**(pid)

INT nproc — The actual number of killed processes.

BOOLEAN val TRUE if the specified process was killed; otherwise False.

AID arobj-id The unique id of the arobject instance.

PID pid The process id of the target process.

GKILL-OP options The options indicate various control options. For example, it can indicate whether the caller stops every time after killing a single process or not.

On Error: An error condition, such as non-existent arobject id or pid, will be noted, and detailed information regarding the error status will be available by looking at the caller's error block. If the target arobject or process was already killed, then no action will be performed and "FALSE" will be returned.

4.3.11.4 Monitor Message Communication Activities for Arobject/Process

The *CaptureComm* primitives capture on-going communication messages from the specified arobject or process. A *CaptureCommArobject* primitive captures all of the incoming request and outgoing reply messages to a specified arobject and can select a target message based on the name of the operation. A *CaptureCommProcess* primitive captures all of the incoming messages and outgoing reply messages for

The *WatchComm* primitives are similar to CaptureComm primitives except that all of the monitored messages are duplicated, not captured.

---

val = **CaptureCommArobject**(arobj-id, commtype, requestor, opr)
val = **CaptureCommProcess**(pid, opr)
val = **WatchCommArobject**(arobj-id, commtype, requestor, opr)
val = **WatchCommProcess**(pid, opr)

BOOLEAN val TRUE if the monitoring action was initiated successfully; otherwise FALSE.

AID arobj-id The unique id of the arobject instance.

PID pid The process id of the target process.

MSG-Q commtype This indicates either "REQUEST" or "REPLY" type.

AID requestor The aid of the communicating arobject.

OPR-SELECTOR opr

The operation to be performed. The "opr" parameter can be a specific operation name or "ANYOPR".

On Error: An error condition, such as non-existent arobject id or pid, will be noted, and detailed information regarding the error status will be available by looking at the caller's error block.

4.3.11.5 SetErrorBlock

A *SetErrorBlock* primitive sets an error block in a process's address space. A user error block consists of a head pointer and a circular queue. The head pointer contains a pointer to an entry which contains the latest error information in the circular queue. After the execution of this primitive, a client can access the detailed error information from the specified error block.

It should be noted that the *SetErrorBlock* primitive will be executed at the process creation time (in the library routine), so that the system default error block will be set automatically.

---

val = **SetErrorStack**(errblock, blocksize)

BOOLEAN val TRUE if the error block is set successfully; otherwise FALSE.

ERROR-BLOCK *errblock
The address of error block.

INT blocksize The size of error block in bytes.

On Error: If the address of the error block is not valid, then the primitive fails and returns FALSE.

4.4 Rationale for the ArchOS Client Interface

This chapter is organized in parallel with the organization of the first chapters of this document. A rational approach to reading this chapter would be to remove this chapter from the document placing it side by side with the remaining sections of the document and then reading it in parallel with the points made in the remaining sections.

4.4.1 Introduction

The preceding sections describe in some detail the specifications for the ArchOS client interface. Most specification documents would end here having as completely as possible specified the operations and the expected responses of the operating system. This specification, and others like it, however, embody the results of a large number of decisions. This chapter is designed to describe the rationale for the decisions made. Obviously not every decision can be completely described here. It's entirely possible that there will be a number of important decisions which we will not describe, but our attempt is to describe all those trade-offs that we have consciously made with respect to the overall functions involved. We will try to identify alternative configurations that we had discussed and will try to identify the reasons for the particular decisions made.

4.4.2 ArchOS Computational Model

The rationale for this section must perhaps be the most incomplete, since there are of course, an large number of choices one can make for the computational model of a distributed system. Distributed systems have been built using models varying from systems built on a star configuration where one node is completely in charge of the system and the others operate in a slave relationship, to autonomous systems; networks of multiple processors tied together and communicating to solve either a common problem or a large set of disjoint problems. The purpose of ArchOS is to build, as we have stated, a distributed computer (i.e., a set of processing nodes which, operating together, act as a single functional entity to solve a particular application problem). Thus we needed a computational model that would reflect the unity of purpose inherent in such a concept.

In addition, we were concerned with the software engineering aspects of the application design. In many existing real time systems (uniprocessors as well as multiple processor systems) we find that there is a strong tendency to design the application software along the lines of the operating system interface, thus causing application partitioning to occur in the program at points that do not correspond to the application problem itself. This effect is perhaps most clearly illustrated with a standard Navy real time operating system in which each event must be handled by a user process specifically designed to handle the event. Thus a single process may not handle both time and I/O events, and sequential I/O operations must be performed by separate process invocations, resulting in a very disjoint (and non-modular) program structure. This creates a problem with the reliability and maintainability of the application system. Although ArchOS is not a production system, we expect that eventually a production system will be built along the lines explored by this Archons research and therefore we would like to start with a computational model which will lend itself to good software engineering practices. In today's technology we felt this included first of all the need to define the

application as instantiations of abstract data types, but here we wanted to ensure that the abstract data types we produced could exploit our distributed environment.

It is also true that in existing real time systems, and particularly in command and control systems such as we are considering for our target applications, the programs are frequently very large and are constructed by large teams of programmers. We would like to have a computational model which would allow not only good software engineering practices with respect to small programs (i.e., programming in the small), but also with respect to the problems of building software in the large. Hence, we have chosen a large primary entity for our computational model, the arobject. The size of the arobject and its clean interface lends itself to a reasonable organization of software engineers for development purposes. In addition, the arobject to arobject interface from is sufficiently simple to allow a software engineering organizational break-out along arobject lines. This is intended to simplify not only the software development of a highly modular application, but also the test and evaluation phase of such a large system.

4.4.2.1 Principal Components

Clearly, our choice for the top level principal component (i.e., the distributed program), is a natural one in light of the fact that the Archons project is producing a distributed computer and an operating system for that computer. We see the distributed program as being a single entity with respect to its overall function, although it is made up of a number of much smaller components. We have chosen not to be concerned with the execution of more than one distributed program, although we have not prohibited it since more than one distributed program may need to be present during application testing. It may be argued that we are merely playing with words here, and in a sense we are, but the only distinction between a single distributed program and more than one is that resource allocation priorities between arobjects in the distributed program are defined, but resource allocation priorities between arobjects in different distributed programs cannot be determined. So it is possible to take two programs which are disjoint and merge them together by defining these interrelationships. Clearly, although we do plan to be able to operate ArchOS in an environment with more that one program running, we cannot claim that resource management decisions will be made fairly between the two systems.

As we have stated, the arobject is a distributed abstract data type. We have taken from the Ada model the concept of a separated specification and body. Similarly, the specification section is the only portion which is visible in the computational sense from one arobject to another. We see the arobject as a distributed abstract data type which can be instantiated more than once in the distributed system.

We should note that an instance need not be resident on a single node. We felt that tying the instance of an arobject to a given node would render inflexible the potential use of an arobject to handle a distributed abstract data type. The decision to limit objects to a single node has been made in the design of a number of systems, such as the Eden system [Almes 83], in which an Eject (conceptually somewhat similar to an arobject) must be entirely resident on a single node. Obviously, the decision to be resident on a single node has the advantage that a common address space can be used for the entire object. We felt that not requiring the entire arobject address space be contained on a single node would give us great flexibility in application design, so we deliberately chose not to require such an organization.

By allowing more than one instance of a given arobject, thus to be bound to a single reference name, we have made it possible for a higher level arobject to completely handle a distributed abstract data type. One could envision, for example, a partially replicated directory existing on a number of nodes or possibly the entire set of nodes in a distributed system, but being handled by a single arobject distributed over that network. The specification portion identifies the entire interface to such a group of arobjects by external arobjects, but processes within the arobject itself can communicate with each other across the various nodes regardless of the node boundaries. Obviously, the performance of such a system must be taken into account and the arobject will have options at its instantiation time with respect to ensuring that particular subcomponents (processes, private abstract data type instances, and so on) are resident on common nodes or separate nodes as dictated by its requirements.

The ArchOS file system illustrates some of the arobject lifetime CHARACTERISTICS. We have chosen at this point to make very simple our concept of the file system by simply using permanent instances of file arobjects [Almes 83]. Such arobjects can be instantiated or killed as required and the set of their permanent arobject id's then becomes, in effect, a directory of the files being kept on the system. The operations of these arobjects would provide the primitives required to access, modify, update and delete the data within the arobject. We would envision, however, that not all arobjects would be permanently instantiated. In fact, quite a number of them might be instantiated during the system start-up operation and would die automatically when the program is terminated for any reason, such as by powering off the system. Thus, an arobject instance might never have a permanent form in existence.

The arobject body is intended to implement the operations described in the arobject specification. It should be noted that the private abstract data types comprise the only form of shared data between processes within an arobject. These are classical abstract data types which contain the data itself

and which can be manipulated only via the defined procedures in the abstract data type. Encapsulating atomic or permanent data within a private abstract data type allows controlled access to shared data. ArchOS will require that any procedures which access atomic data must have a transaction open at the time of access. Thus, ArchOS can control the sharing of this data, and when the transaction is committed or aborted, ArchOS will ensure that the atomic data is correctly forced to appropriate stable storage. We also note that because of our arobject distribution provisions, the private data can be partitioned among multiple nodes, although each individual private abstract data type instantiation must be fully resident on a single node. In this way remote procedure call semantics, including parameter passing by value, will be used if the process is not co-located with the private procedure on the same node. It is intended that defining the private abstract data types in this way will allow us to directly implement such distributed applications as a partially replicated directory and in fact, an example of such an implementation can be found in Appendix A of this document.

At this point, let us observe that there is no required correlation between the operations of an arobject and the processes defined in an arobject body. It's anticipated that one or more processes in each arobject body will run continuously, pausing from time to time to check for the existence of an operation request from another arobject. If such a request is not found, that process could block awaiting another such request. Thus, the operation can be handled by any process within the arobject, which allows us to have a set of processes with essentially identical function, handling operations in any manner that the application desires. The binding between these operations, then, is late and is handled dynamically by the appropriate processes. If, of course, the designer wants to couple the processes to the operations he need only specify his *Accept* parameters to identify which operations he wishes to accept at each point and therefore he can bind them as early as he wishes. A cost of this approach is that the potential implementation error of omitting the handling of some operation in an arobject cannot be detected at system generation time, but we feel that the benefit in terms of modularity and concurrency greatly outweighs this problem.

Processes within an arobject, of course, may communicate by using the *CreateProcess* primitive to create processes and passing parameters at that time, or they may invoke operations within the arobject using a *Request* primitive, using either operations from the specification part, or private operations from the body. These private operations are designed to make it possible to place operation requests into the incoming queue from internal processing which are separable from those placed by external arobjects. Obviously processes within an arobject can also communicate via shared data, using the private abstract data types. Any scheduling or mutual exclusion which must be handled with respect to this shared data would be handled within the abstract data types.

In addition, we have allowed for inclusion of private arobjects to be defined within the body of an arobject. Such a private arobject will not be visible to external arobjects, and could be used for partitioning operations within an abstract data type. This technique could be used for an implementation of a partially replicated directory by placing each partition in its own arobject and using the outer arobject to distribute the lookup and scheduling for the arobjects containing the data. (See Appendix A Solution 2.)

One may, of course, question our contention that these processes are lightweight as we have defined them; that their scheduling overhead will be small relative to the speed of the machine. This is certainly our intention, but we will have to weigh this against the implementation requirements to obtain the interfaces we need. It is our intention, however, to minimize the state required to be constructed in the scheduling of a process. In addition, we envision eventually building special purpose hardware for hosting ArchOS, one objective of which will be to optimize the creation of processes and the resulting context swap overhead.

4.4.2.2 Communication Facilities

4.4.2.2.1 Rationale for Accept/Request Rendezvous Mechanism

The *Request/Accept/Reply* primitives were selected to provide the means of communication among arobjects. These primitives allow communicating arobjects to rendezvous, rather than providing a master/slave relationship for all communications.

The master/slave paradigm was rejected because it seemed too restrictive. Consider that a major goal of ArchOS is to support decentralized resource management by collections of resource managers, which negotiate to reach a consensus regarding a particular management decision. The communications involved in carrying out negotiations among resource managers are not master/slave in nature; they are better characterized as communications among peers.

The *Request/Accept* rendezvous captured this sense of peer communication--in order for communication to take place, both parties involved must explicitly act; the requestor cannot force a process to perform an *Accept* for a particular invocation.

Both blocking and non-blocking request primitives are provided to give the client a very flexible communication facility.

4.4.2.2.2 Rationale for Broadcast Request Capability (*RequestAll* Primitive)

The *RequestAll* primitive provides the capability to broadcast a request to multiple arobjects. This feature of the inter-arobject communication facility was very directly shaped by the anticipated structure of ArchOS internal distributed data entities.

It is assumed that the ArchOS operating system will make use of various data objects that must be highly reliable and available. Such objects can be constructed by means of data replication and distribution throughout the operating system, with appropriate use of the ArchOS transaction facilities. In addition, ArchOS will almost certainly contain multiple instances of various servers (for instance, file servers and name servers). If data is either partially or completely replicated at multiple locations or if replicated servers are available at multiple locations, it seems natural to use broadcasts as a common mode of communication involving these replicated entities. For example, a new entry in a name table might be broadcast to all of the arobjects that contain a portion of that name table. Each partially redundant table fragment could then be updated appropriately.

4.4.2.2.3 Rationale for Intra-Arobject Request Capability

Despite the number of communication primitives provided by ArchOS, we wanted to keep the model of communication among entities as uniform as possible. As a consequence, ArchOS does not support a direct process-to-process communication capability. All processes communicate only with arobjects, by means of the Request/Accept mechanism, even when a process wishes to communicate with another process in the same arobject. (This restriction is a consequence of the fact that we do not want the user of an arobject to know about the implementation of another arobject (including the number of processes and the process id's in that other arobject). The arobject operation invoker can only see the specification, not the implementation, of the arobject to be invoked. Therefore, process-to-process communication between two distinct arobject instances is not possible. And in the interests of uniformity and simplicity, we also decided not to allow such communications to take place within a single arobject instance.)

In order to allow processes in a single arobject to perform additional operations, beyond than those that are visible to other arobjects, it was necessary to provide an additional set of operations that can only be invoked by the processes in that arobject. These operations are called the *private operations* of the arobject, and they are invoked in exactly the same manner as operations on other arobjects.

4.4.2.2.4 Rationale for Invocation Parameter Passing

ArchOS *Request* parameters are passed to the receiving arobject using call-by-value semantics. This is the only reasonable way to pass parameters since arobjects do not share any address space.

The only exception to this rule comes in the case of an invocation by a process of a private operation. In that case, the processes can have intersecting address spaces due to the presence of shared private data (in private abstract data type instances) in the arobject. As a result, it is possible for such *Request* and *Reply* messages to contain references to common objects (for example, the names of private data type instances).

4.4.2.3 System Load and Initialization

The tradeoffs involved in this section are fairly simple. We expect that the system would be loaded in a conventional manner from external storage (e.g. disk) associated with some of the nodes in the system, and we expect that nodes not containing local external storage would obtain load data from neighboring nodes. We expect this to be an internal ArchOS design decision, which will therefore be specified at a later time.

The question to be answered in this section is how the application program would be initialized once ArchOS is fully initialized and has determined its own status. We have taken the position at this point that an application program will do this by the definition of a unique application arobject (the *Root* arobject), which ArchOS will expect to find on one or more nodes. This arobject will then determine its own status, including finding out if other copies of itself exist (or insuring its own uniqueness if needed). This arobject will then *Create* the other arobjects needed to bring up the application program.

This is a simple approach which has been planned to maximize application program flexibility at initialization time, and to eliminate the need for operator intervention other than that required by the application itself.

4.4.2.4 Transactions

ArchOS is a highly decentralized, real time operating system designed to support highly decentralized, real time applications. In fact, the operating system itself can be viewed as a highly decentralized, real time application built on top of the kernel support facilities. We have attempted to view ArchOS in this way at various times, and that has led us to view the arobject as a computational entity that we would use *within* ArchOS (insofar as possible), as well as at the application level. While specifying the services to be provided by ArchOS and its behavior under all conditions, it became

apparent that ArchOS' internal system data objects should possess several characteristics. In particular, the following characteristics were desired:

- Often, several different arobjects will operate on specific data items (directories, queues, and so on). These shared data objects will reside in an arobject, so access can be coordinated by the normal arobject communication facilities (particularly *Accept* primitives), as well as by means of customized code in the arobject's processes. But, as the following points will illustrate, a higher level coordination will often be desirable.

- At times, it is necessary to change several different, yet related, data items as a unit (for example, updating all of the copies of an entry in a partially replicated directory or moving an element from one queue to another). If such atomic updates could be performed, then it would be much easier to transform one consistent state of a set of data items to another consistent state.

- Some data items must be permanent; that is, their state should be reliably maintained for the life of the system.

These attributes could all be provided by ArchOS by means of the communication facilities in conjunction with custom-written code and appropriately defined locks or semaphores and critical regions. However, we desired a more structured approach. All of the above capabilities are supported by traditional database transaction systems [Gray 77]. In fact, such transaction systems can provide even more powerful properties (specifically, failure atomicity and/or serializability). It was felt that this additional structure would ease the programmer's burden, while also decreasing the chances for programming errors, by making modular programming more natural. (Indeed, the use of compound transactions promotes modular construction of programs by causing the transaction author to think in terms of consistency preserving transformations on sets of atomic data objects, thereby causing the program's data to be partitioned into a number of modular atomic data sets. Also, transaction systems in general can aid the programmer in another important way: the processing carried out in order to commit or abort a transaction can handle a great deal of lock-related bookkeeping, even though the programmer must explicitly obtain locks on all of the atomic data items. This frees the programmer to consider the correct behavior of the transactions being written, without giving unnecessary consideration to interactions with other transactions in the system. However, this is not to say that the programmer does not have to be concerned at all with locks and locking protocols; rather, it is intended to point out one aspect of lock management that the programmer does not have to handle. Using the current ArchOS locking protocols, there are still a number of decisions concerning locks that the programmer must make--for instance, whether to use a discrete locking protocol or a tree locking protocol, how to organize the locks in a lock tree, what data items are associated with a given lock, and so on. Section 4.4.2.7 deals with some of these issues in more detail.)

The above discussion explains why a transaction facility was included for use *within* ArchOS. Since the ArchOS clients are also interested in producing highly decentralized, real time programs, it was felt that it was appropriate to extend these primitives to the clients as well.

Of course, there are arguments against using a transaction facility within an operating system. One major objection is that system performance could be greatly reduced (as compared to a system that does not use a low-level transaction facility). This is due to the fact that occasionally the system will have to suspend the processing of a specific transaction while data is being copied from main memory to a (virtually) permanent medium; there will also be overhead associated with the initiation and conclusion of each transaction. (The impact of mutual exclusion on system performance is not mentioned in the preceding discussion since some form of mutual exclusion must be present in any system containing shared data objects, whether it includes transactions or not.)

It seems inevitable that a performance penalty will be incurred by the use of transactions, but it is hoped that the gains in the area of data permanence, system consistency, high availability, and reliability will be worth the price. However, three other decisions were made to address the problem of performance losses due to the use of transactions in ArchOS:

- not all processing must take place within transactions.
- only specifically designated data items would be defined as *atomic*--that is, only those items would be permanent, failure atomic, and so forth. (This decision forces the arobject writer to explicitly indicate which data items must be atomic and has a great influence on the types of steps that appear within a transaction. For instance, it would be a questionable, if not wrong, programming practice to use non-atomic variables to pass values from one nested transaction to another. An example illustrating this point is shown in Section 4.4.3.5.)
- a new type of transaction, the compound transaction, is included to increase the potential degree of system concurrency.

4.4.2.5 Rationale for the Inclusion of Compound Transactions

Compound transactions addressed two great concerns regarding the use of transactions, both within ArchOS and by ArchOS clients. One of these concerns, performance, has already been mentioned. Since ArchOS allows transactions to be nested arbitrarily (interleaving elementary and compound transactions as desired), the use of some compound transactions can increase system concurrency. This is due to the fact that compound transactions release all of the locks that they, or any of their child transactions, have set at the completion of the execution of the compound transaction. Thus, the resources that are controlled by these locks are often free to be used by other processes prior to the completion of all of the processing associated with a given transaction.

The second concern addressed by compound transactions in ArchOS is that of system integrity and liveliness. In traditional database systems which support nested transactions, all of the locks obtained by child (nested) transactions are passed to their parent transactions and kept until the completion of the highest level transaction (at which time the transaction is either aborted or committed). This approach was not suitable for ArchOS, where most of the operating system primitives are actually expected to be implemented using transactions. If a client transaction contained operating system primitive calls which were implemented using traditional nested transactions, then on the completion of the primitive call, the client transaction would receive any locks that the system primitive transaction(s) had obtained for system resources. The client could subsequently attempt to manipulate the system resource or could simply hold the lock on the system resource for an arbitrarily long time. Both of these possibilities were disturbing; but, both were also preventable by proper use of compound transactions. If each ArchOS primitive is not just a transaction, but rather a compound transaction, then no locks on system resources will ever be returned to the client transaction. In this way, compound transactions are used in ArchOS to build a "firewall" between the operating system and the client.

Of course, compound transactions have some disadvantages associated with them as well. For instance, it is not possible to simply change an arbitrary elementary transaction to a compound transaction without consideration of recovery issues. (Elementary transactions in ArchOS correspond to traditional nested transactions in database systems.) During the course of its execution, each compound transaction may invoke a number of arobject operations and/or operating system primitives. At the completion of the execution of the compound transaction, all of the locks that have been obtained during transaction processing are released (despite the fact that the compound transaction may be nested within another transaction). In the event that a higher-level transaction aborts after the compound transaction has committed, it is not possible to guarantee that all of the operations performed by the compound transaction can be properly "undone" (in the sense of traditional nested transactions). For instance, it is possible that another transaction has read, and acted upon, data that represented the outcome of the compound transaction after it had committed (and thereby released all of its locks), but prior to the execution of its compensation action. Such a situation could never arise in a traditional transaction system, but it certainly could happen in the ArchOS transaction system.

The ArchOS compensation action for a committed compound transaction consists, in part, of the execution of a set of compensation operations associated with the arobject operation invocations and operating system primitive invocations made during the course of execution of the compound transaction. While these compensation operations may attempt to approximate the effects of the

traditional transaction "undo" operations, they cannot guarantee that the compensation will result in the same system state as would have resulted if only nested elementary transactions been used. Rather, the system state is transformed to a state that is equivalent to the state that would have resulted if all of the other concurrent transactions in the system had been processed in the absence of the aborted compound transaction. (See Section 4.2.4.1 for additional discussion of this point.) If such compensation operations can be constructed and the weaker guarantees concerning the system state in the case of the abortion of a higher-level transaction are acceptable to a transaction author, then compound transactions can be used for a given application; however, if these conditions are not sufficient, then elementary transactions must be used.

A few examples can be used to demonstrate cases in which compound transactions are or are not appropriate based on the ability of compensation actions to provide the required semantics.

First, consider a case in which compound transactions are appropriate: the dequeue operation of a weak queue. In such a queue, the first-in-first-out ordering of elements in a strong queue is weakened; it is acceptable to alter the order in which queue elements are removed from the queue by the dequeue operation. As a result, it is possible to dequeue elements by means of a dequeue operation based on a compound transaction. This operation simply returns the head element of the queue and then releases any locks obtained in the process of dequeuing that element. The compensation operation associated with this dequeue operation is also quite simple: the element that was previously dequeued is returned to the head of the queue. Because of the semantics of the weak queue, these operations are acceptable. It is unimportant whether or not any other dequeue operations occurred during the interval between an element being dequeued and subsequently being requeued because a strict ordering of the queue elements is not required.

Second, consider a case in which a compound transaction is inappropriate: the enqueue operation of a weak queue. In this case, there is a visibility problem--that is, if compound transactions were used to implement the enqueue operation on a weak queue, it would be possible for other transactions to view the queue in states that represent partial results of computations that are subsequently aborted. As a result of viewing such states, it is possible that those transactions will alter the state in an inappropriate way. This situation is illustrated by the use of a compound transaction to implement the enqueue operation for a weak queue. Such an enqueue operation would take an element passed to it and append it to the tail of the queue, releasing any locks obtained at the completion of the operation. The most obvious compensation operation for this enqueue operation would locate the desired element in the queue and remove it, thereby attempting to make it appear as though it had never been there. However, this is not a sufficient compensation action since

it is possible that an element may be enqueued and later dequeued before the compensation action is able to be executed. In such a situation, the only way to provide the required semantics for the weak queue is to abort the transaction that dequeued the element. Yet this presents the possibility of cascading aborts, and ArchOS cannot permit cascading aborts to occur. Due to this visibility problem, compound transactions are not appropriate for the implementation of the enqueue operation.

Another disadvantage associated with the use of the compound transaction is also related to the compensation mechanism: programmers must explicitly write the compensation operations corresponding to the arobject operations for a given arobject. The concept of this type of transaction is quite new, and we are not yet certain about the nature of the actions to be performed by a typical compensation routine. (In fact, there are many issues that we do not fully understand with respect to compound transactions: the number and variety of applications for compound transactions, the amount of work required to define appropriate compensation actions, the form such actions should take, the impact of compensation actions on the ability of ArchOS to make guarantees about real time behavior, the level of concurrency that can be achieved using compound transactions, and so on.) So at this point, we have decided that ArchOS will initially support only programmer-coded compensation actions. (This should be contrasted with the case of traditional nested transaction database systems or, equivalently, nested ArchOS elementary transactions. In these cases, all of the "undo" or "redo" types of operations are determined and performed by the transaction facility for any user-written transaction.) However, ArchOS will provide support to automatically compose these basic compensation actions in order to facilitate the construction of arbitrary higher-level transactions (either compound or elementary transactions) that invoke arobject operations based on lower-level compound transactions.

4.4.2.6 Rationale for the Transaction Syntax

Two potential formats were considered for the syntactic definition of transactions: an in-line format (in which the transaction would be delimited in the body of the surrounding text by some keywords) and a procedural format (in which a transaction was defined as a separate entity--such as a function or procedure--and was "called" by the transaction initiator).

There was no overwhelming reason for choosing one format over the other. This issue seemed to be largely one of stylistic preference, not performance or functionality. Assuming that a typical transaction is relatively short, the in-line format has the advantage of showing the actual transaction steps to a reader of the code; on the other hand, the procedural format could be parameterized in the hope of avoiding the duplication of definitions that might occur with the in-line format. (This seems to

parallel the arguments for using macros or subroutines in a given application.) Our preference was to use the in-line format since it appeared to be more readable and compact.

Once the determination to use an in-line format was made, we needed to pick a specific format. We felt strongly that transactions should not span multiple arobjects--that is, a transaction should not be initiated by one arobject and later completed by another arobject. Since the arobject is the basic unit of program construction, transactions that span arobjects hardly seem to promote modular program construction techniques. In fact, we felt that a transaction should begin and end in a single process. The reasoning for this decision is a simple extension of the argument previously given for requiring the transaction to begin and end in a single arobject. The *ET{...}* and *CT{...}* syntactic structures selected for use in ArchOS force a transaction to begin and end in a single process, while other possible structures (such as arbitrarily placed *BeginTransaction* and *EndTransaction* delimiters) did not.

4.4.2.7 Rationale for Lock Support Decisions

Several important decisions were made with respect to the support to be provided to the client in terms of obtaining locks on shared data objects. This portion of the rationale will deal with three of the most important decisions: (1) the decision to support both a discrete locking protocol and a tree locking protocol [Silberschatz 80]; (2) the decision that the client must explicitly set locks ; and (3) the decision to allow the client to explicitly release locks within a transaction.

ArchOS supports both discrete locks and tree locks because we feel that each type of lock addresses a different set of client needs, and neither type alone addresses all of these needs. For instance, the tree lock is supported because it is able to make an important guarantee about the nature of the computations that use exclusively tree locks from a single lock tree: if a computation, C, obeys the tree lock accessing rules and if all of the other computations that obtain locks in that tree release them in a finite length of time, then computation C will also complete in finite time *without the occurance of deadlocks*. We believe that the guarantee that a computation will take place without the possibility of a deadlock is extremely important and justifies the support of tree locks in ArchOS.

However, tree locks cannot be the only locking mechanism in ArchOS. In defining the tree structure to be used in connection with tree locks, the client is explicitly specifying the legal access patterns for locks in the tree. This may be a straightforward process when the client is dealing with a small collection of related data items, but appears to be intractable when dealing with all of the locks in the system. (If it were desired to guarantee that the entire system would be deadlock-free, then it would be necessary to place all of the locks in the system in a single lock tree. Specifying a rational tree

structure for such a large number of often loosely related items seems impossible.) This leads us to support discrete locks as well as tree locks. (Actually, ArchOS will provide support for multiple lock trees defined by the client, as necessary.)

In fact, we could build ArchOS without discrete locks, using a forest of tree locks. Yet, once it was decided that we could not include all of the locks in a single monolithic tree, it seemed to be worthwhile to allow the more traditional discrete lock to be used as well. Presenting the view to the client that each discrete lock is actually a degenerate (single node) tree lock seemed to be unnecessarily complicated. (Although that may be the manner in which discrete locks are actually implemented.)

The next major point to be discussed is the justification for the rule that a client must explicitly obtain all of the locks needed to perform a given computation. This decision was reached for two reasons. First, a system that would handle the acquisition of locks automatically would be at or beyond the state-of-the-art for database systems. Since this is not related to ArchOS' prime research goals, we would prefer not to expend the effort that such a capability would require. Second, even if we had an automatic lock acquisition facility, it seems inevitable that the system would be more prone to deadlocks than if the client explicitly requested the locks. This is due to the fact that the automatic system would often have to upgrade a "read" lock to a "write" lock during the course of a transaction. The attempt to upgrade the lock could lead to a deadlock situation, which might have been avoided if a "write" lock had been requested in the first place. Although the automatic system would have no way of knowing that a "write" lock would eventually be needed, the client who wrote the program would indeed have known and could have avoided the situation in many cases.

The final major point to be addressed concerning locks is the use of the *ReleaseLock* primitive--specifically, a justification for the use of this primitive within a transaction.

Although the use of the *ReleaseLock* primitive within a transaction can violate the locking protocols of the transaction system, it can also be used in a manner consistent with those rules. In particular, tree locks can be released during the course of a transaction if they are not needed for the transaction computation. For example, if a given tree lock were obtained only to lock some of its descendants in the tree, then that lock could be released after the locks on the descendants have been obtained, with no undesirable effects. Also, it is possible that explicit releasing of locks might be useful in taking full advantage of Sha's notion of setwise serializability [Sha 84].

A few other minor items concerning locking issues should be mentioned. The syntax chosen for

declaring locks in an arobject is similar to the syntax used in declaring variables (with types DISCRETE – LOCK – ID or TREE – LOCK – ID). However, the tree structure of the tree locks is defined dynamically by means of the *CreateLock* and *DeleteLock* primitives.

Also, we have not yet made a final decision concerning the "strength" of the connection between a lock and the data item it is associated with. If locks are closely associated with the data item, then the system can perform a number of checks to guarantee that the locks are being used properly. However, if this bond is weaker, then the client has more freedom to associate locks with more general or more abstract data items or even facilities. (For example, the client could obtain a lock on an arbitrary character string which may not correspond to any data item at the time the lock is obtained. This capability might be more difficult to provide if locks are tightly associated only with existent data items.) In this case, though, the client must use a programming convention in order to guarantee that the locks are used properly in accessing the data. These two examples are the end points of a range of possible lock strengths. We have yet to decide where in this range of possibilities the ArchOS supported locks should lie.

Finally, ArchOS does provide a client-specified timeout parameter to bound the execution time of a given transaction. This provides a crude facility to prevent deadlocks from tying up the transaction for an arbitrarily long time. ArchOS will not necessarily detect a deadlock condition for the client; the ultimate responsibility for handling the possibility of a deadlock belongs to the client. However, ArchOS will provide some deadlock detection facilities. The exact nature of these facilities has not yet been fully determined, but some candidate facilities are: ArchOS will detect all deadlock cycles of length two (or perhaps three), or ArchOS will detect all deadlocks that involve the resources of only a single node. The level of deadlock detection service provided will be strongly influenced by the cost of providing that service. Since we will not be able to detect all deadlock conditions, we will only perform that deadlock analysis that is relatively inexpensive, while still capable of detecting some of the more common situations. In the event that a deadlock is detected, ArchOS will abort transactions as required in order to allow processing to continue without deadlock.

4.4.2.8 Rationale for Inclusion of Critical Regions in ArchOS

ArchOS supports critical regions to assure exclusive access to shared data items within a single arobject by means of the *Region* primitive. However, it may be noted that ArchOS also supports another, more secure, method of obtaining mutually exclusive access to shared data items: the transaction.

Since ArchOS is already committed to providing a transaction facility, it may not be obvious why

critical regions are also supported. In fact, the main reason is that critical regions do not require all of the powerful facilities that a transaction supplies, whether they are needed or not. It was decided that no matter how efficient the ArchOS transaction facility was, it would still probably be slower than a mechanism that only provides a small fraction of the power of a transaction (even a compound transaction, which would typically involve less processing than an elementary transaction). As a result, the critical region can be used to provide a simple mutual exclusion mechanism that will often be needed when accessing shared data objects for which the failure atomic, permanent, or serializable properties of transactions are not required.

4.4.2.9 Rationale for Transaction Nesting Rules

ArchOS allows both elementary and compound transactions to be nested arbitrarily within a transaction. Such interleavings are necessary for proper system behavior. This will be demonstrated by a number of examples that point out the necessity of each type of nesting:

- In order to provide the traditional database nested transaction view, it is necessary to allow the nesting of elementary transactions within elementary transactions.

- Where failure atomicity (the property by which either all of the actions of a transaction are performed or none are) and visibility issues (as mentioned in Section 4.4.2.5) are not of prime concern, system performance can be maximized by employing nested compound transactions rather than nested elementary transactions.

- In cases where compensate routines can provide similar functionality to more traditional "undo" operations on locked data items (for example, consider the case of compensating for the allocation of a new page of memory from the operating system), it would be advantageous for performance reasons to use compound transactions nested within elementary transactions.

- Consider the case in which a compound transaction must perform a certain computation. It is certainly reasonable to believe that it might often be carried out by a set of nested elementary transactions, thereby giving all of the automatic failure recovery and visibility features of nested elementary transactions to the desired computation. It is of no interest to the lower nested levels that their locks will all be released as soon as the compound transaction commits (or aborts). (Note that the compound transaction in this case acts a great deal like the top-level transaction of a set of nested elementary transactions.)

Since all of these cases seemed to be useful, it was established that ArchOS transactions could be constructed by any arbitrary mixture of the two types of transactions.

Such mixtures of the two transaction types can be quite useful in selectively passing the locks of some child transactions to a parent transaction while releasing those of other child transactions. This ability to selectively pass locks is required since the ArchOS primitives will usually contain compound transactions (in order to prevent passing system resource locks to clients), yet client elementary

transactions may be built by making *Requests* for services from other client arobjects. If a *Request* were simply a compound transaction, and the computation that resulted from the *Request* execution were considered to be a child of the *Request* transaction, then none of the requestee's locks would be returned to the requestor (client) due to the nature of a compound transaction.

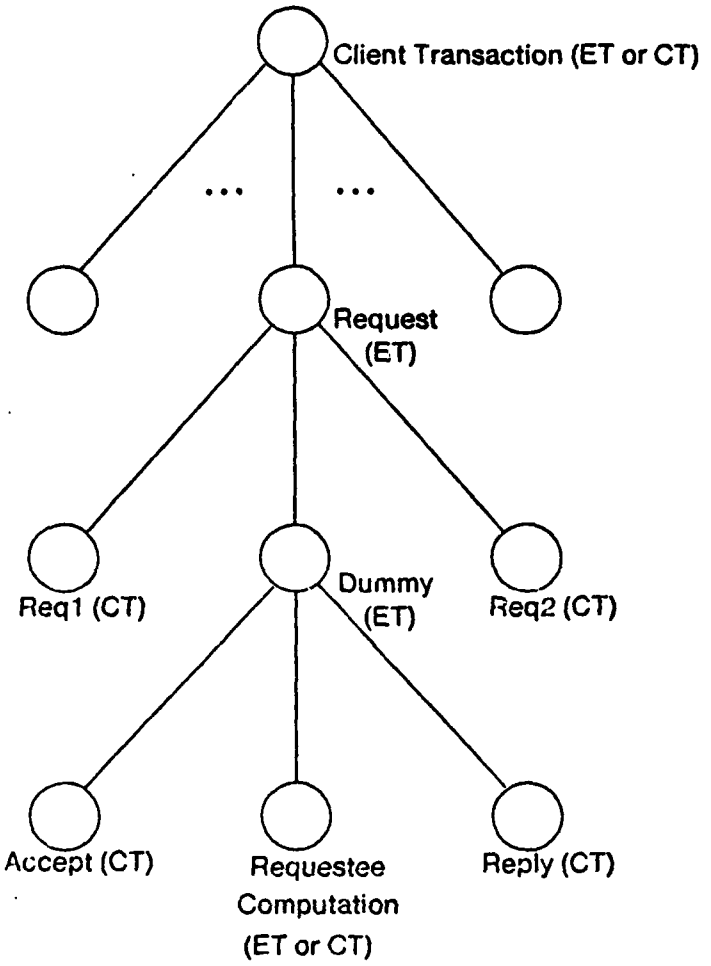


Figure 4-4: Selective Lock Passing by the *Request* Primitive

A mixture of elementary and compound transactions can be used in the above example to pass the requestor the requestee's locks while releasing the system resource locks at the completion of the *Request* primitive execution. (See Figure 4-4.) By implementing the *Request* primitive as an elementary transaction with several child transactions, the desired effect can be achieved. The locks obtained by the requestee's computation are automatically passed back to the requestor; and, the system actions (Req1 and Req2 in the figure) are encapsulated within child compound transactions,

so the requestor will not receive any of the system resource locks (since they are not returned to the *Request* primitive's highest level elementary transaction.)

4.4.2.10 Rationale for Inclusion of the AbortIncompleteTransaction Primitive

The *AbortIncompleteTransaction* primitive provides a unique capability in handling responses from child transactions.

To understand how such a primitive facility could be used, consider the following situation: if a transaction performs a *RequestAll* invocation for some operation on all of the arobject instances of a certain type, then each invocation will be serviced and appropriate replies will be returned. Also, suppose that while servicing the request, each arobject instance performs some transaction processing. And finally, suppose that the requestor is only interested in the first reply to the *RequestAll* invocation. (Although this scenario may sound contrived, it is not. This is exactly the manner in which a client would attempt to obtain the shortest response time from a set of arobjects, all of which are capable of servicing the client's request. The client would simply request that all of the arobjects provide the required service (by means of the *RequestAll* primitive) and then wait until the first reply is obtained. Since, at that point, the actions of the rest of the servers are no longer of interest to the requestor, their actions can be terminated.)

After receiving the first reply, the client could not correctly proceed without the *AbortIncompleteTransaction* primitive. This is due to the fact that the computations carried out by the arobject instances that respond (or would respond) after the first should be aborted (including the transaction processing that they are carrying out). However, the client has no way of knowing the identity of those arobject instances that received the request since the *RequestAll* primitive was used and therefore the exact identities of the acceptors were never explicitly known by the requestor. If the client aborts all of the processing associated with that *RequestAll* primitive, then the work done by the first replier will be lost as well. The *AbortIncompleteTransaction* primitive handles this case since it allows ArchOS to handle all of the bookkeeping associated with the identities of the acceptors and as well as performing the necessary selective aborts.

Figure 4-5 shows the transaction tree structure associated with the *RequestAll* primitive. This figure will illustrate the actions taken by the *AbortIncompleteTransaction* primitive with respect to the transaction tree. The *RequestAll* transaction has several children transactions, one for each of the acceptors of the request for service. At the completion of each server's *Reply*, the subtransaction associated with that server is concluded (committed). The *AbortIncompleteTransaction* primitive aborts all of the child transactions of the named transaction (*RequestAll*, in this case) that have not

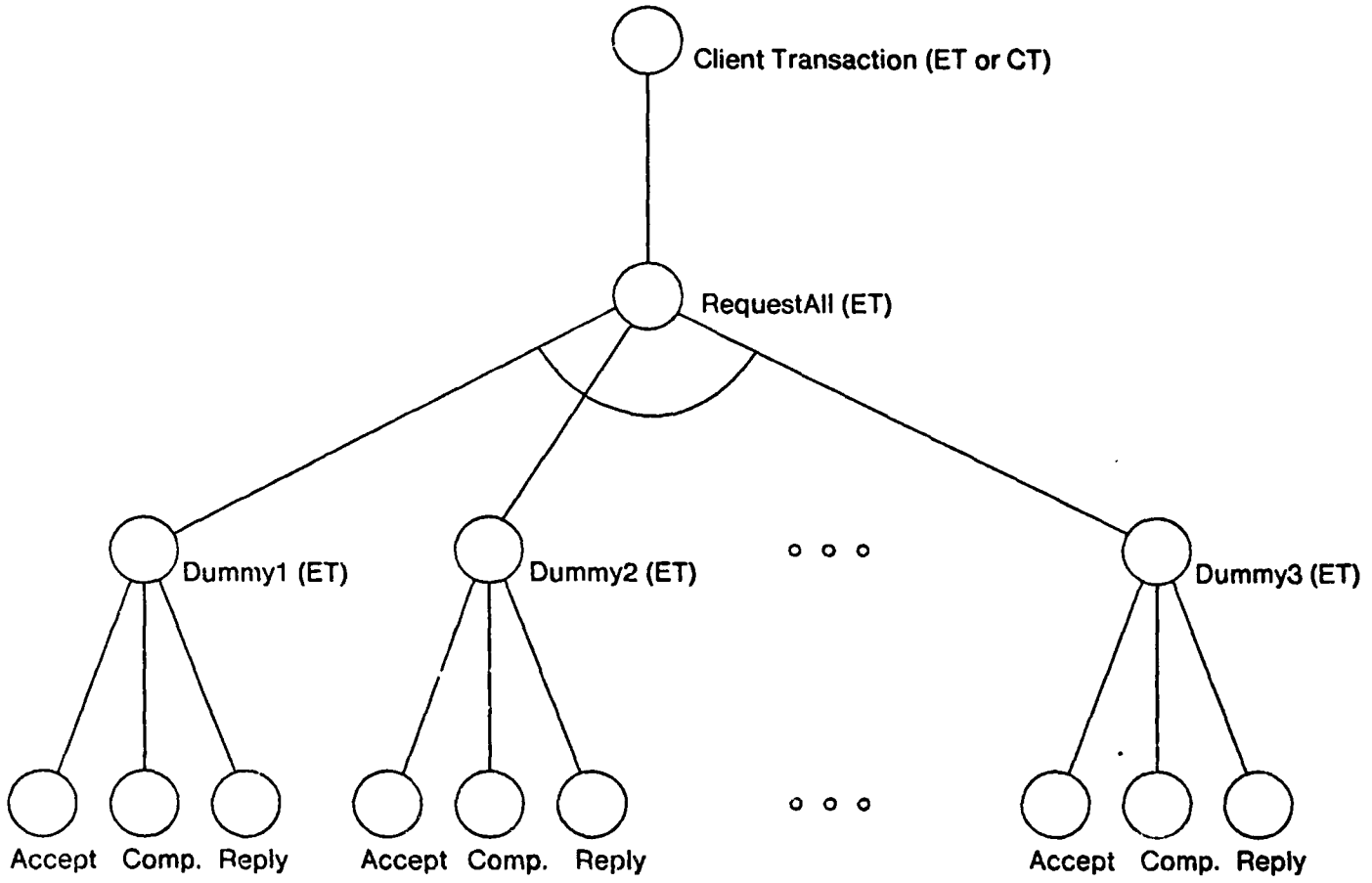


Figure 4-5: *RequestAll* Transaction Tree

yet reached that commit point. In this way, the work that has been done will be maintained, while the incomplete work will be aborted.

This primitive has proven quite useful in example programs, including the partially replicated distributed directory example contained in Appendix A of this document.

4.4.2.11 Real-Time Facilities

ArchOS is designed to support user-defined real time deadlines, but its support for real-time systems differs fundamentally from other real-time operating systems. There are, of course, many existing real-time systems being used in a number of environments including the military, process control, and robotics. It is interesting, however, to observe some of the characteristics found in the operating systems (or *executives*, as they are sometimes called when a full operating system is not implemented) designed for these systems which provide support for real-time operation. Two primary characteristics can be described:

- These operating systems are kept simple, with minimal overhead, but also with minimal function. Virtual storage is almost never provided; file systems are usually either extremely limited or non-existent. I/O support is kept to an absolute minimum. Scheduling is almost always provided by some combination of FIFO (for message handling), priority ordering, or round-robin, with the choice made arbitrarily by the operating system.

- Simple support for management of a hardware real-time clock is provided, with facilities for periodic process scheduling based on the clock, and timed delay primitives.

Conspicuously missing from these systems at the operating system level is any specific support for managing user-defined deadlines, even though meeting such deadlines is the primary characteristic of real-time application requirements. Instead, these systems are designed to meet their deadlines by ensuring that the available resources significantly exceed the actual user requirements, and the implementation is followed by an extensive testing period to verify that this assumption is maintained under "normal" loads. In priority-driven systems, deadlines are handled by assigning a high fixed priority to processes with critical deadlines, disregarding the resulting impact to less critical deadlines.

The ArchOS primitives have been designed to provide the information which will enable the scheduling function to sequence processes according to their deadlines, ensuring that all deadlines are met in each node as long as there are sufficient resources to meet them. Techniques for handling such deadlines are well known, given sufficient resources.

An important ArchOS research area, however, is the handling of these deadlines when there are insufficient resources to handle all of them. User policies defining potential objective functions for these cases will be accepted, and a best effort will be made to implement these policies when resources are insufficient. Essentially two primary decisions are required in these cases:

- Which deadlines should be missed, and by how much? That is, should some deadlines take precedence, or should missed deadlines be fairly distributed? Should some processes be aborted if their deadlines cannot be met?

- At what point should some processes be migrated to other nodes? If so, how many processes, and which processes should be migrated?

4.4.2.12 Policy Definitions

We felt from the outset that an application running under ArchOS would need a great deal of flexibility in order to meet its specifications. Therefore, we decided to support application defined policies to manage certain resources. At this point, we have limited the policy definition capability to two specific management tasks: process scheduling (initially on a single node) and process

reconfiguration (process migration or relocation) among nodes. This decision was made because we did not feel that there was a global programming paradigm involving application-defined policies that we wished to impose on the application program writer; currently, we treat each facility on its own merits with respect to the use of application-defined policies. We do have an overall framework within ArchOS for the support of application-defined policies, but we would like to demonstrate the suitability of our ideas on a few test cases before spreading the approach throughout the entire system.

In addition, process scheduling and process reconfiguration in a distributed system are of particular interest to the real-time application programmer, who is often in the best position to determine the high-level scheduling policies of the system. This is due to the fact that the actual deadline constraints that the system must satisfy are imposed by the external environment and the nature of the application processing. By allowing the application programmer to define the scheduling policy, it is expected that the system will better be able to carry out the process scheduling task with these factors in mind, particularly in situations where there are insufficient processing resources to carry out all of the applications functions.

Another area of interest within the Archons project concerning the use of policy within a facility is the separation of the policy and mechanism portions of the facility. This work is similar in spirit (if not in implementation) to the policy/mechanism separation work done for the Hydra system [Wulf 81]. However, for the most part, ArchOS will not focus much attention on that particular aspect of policy definition for a facility. This is partially due to the fact that, in many cases, it is very difficult to draw the line between those protions of a facility that are policies and those that are mechanisms. And since drawing that line is not a central focus of the ArchOS development effort, we feel that we should not spend a great deal of time pursuing this task. (We will indeed attempt to separate policy and mechanism insofar as possible in ArchOS, but we will not push this idea very hard.)

Once we had decided to provide support for application-defined facility policies and to make an attempt to separate policy and mechanism in the implementation of the facility, there were still a number of open questions.

First, we recognized that there is a spectrum of choices in the amount of flexibility to be provided to the application writer. One approach to the specification of a facility's services would involve: (1) determining the set of all of the policies that might ever be used to manage the facility, (2) selecting a set of mechanisms that can be combined in various ways by that set of policies to provide any of the previously identified facilities, and (3) making those mechanisms available to the application

programmer to construct the facility. While this scheme allows the application writer a great deal of freedom, it is possible that it might be very difficult to actually carry out the first two steps listed above. A second approach limits the options of the application writer (and so might be more secure and reliable from the operating system's point of view) while still allowing the client to select the policy to be used in managing a facility. In this case, the application programmer would be able to select the facility policy from a limited menu of policies. (Essentially, the programmer has been given a policy mode switch.) The programmer would not necessarily even know the mechanisms that actually underlie the policies in this case. For process scheduling when processing resources are not sufficient to meet processing demands, we have chosen a scheme that lies somewhere between the extreme cases listed above. We intend to design a policy scheduling facility that has a fixed set of mechanisms that the client cannot directly access. Some of these mechanisms will evaluate value functions in order to compute an optimal (or near optimal) schedule for processes on a given node. The value functions that correspond to many of the well-known scheduling approaches will be available to the client (in a form similar to a library). However, the client may also specify the parameters for a certain class of value functions and submit a value function to the process scheduling facility to determine the exact policy to be followed. Although this approach does not allow the client to specify an arbitrary value function to the process scheduler, it does allow much more freedom than choosing from a small menu of well-known policies (such as "missing fewest deadlines" or "minimize maximum lateness").

Second, there are a number of places where application-defined policies and policy/mechanism separation can be used. We considered the possibility that we could have two levels of application-defined policies: policies that affected the entire distributed program and policies that affected individual arobjects. We believe that there are cases where both of these alternatives make sense. However, neither process scheduling nor process reconfiguration seems to have a compelling use for arobject-level policies. As a result, the policy definition capabilities described in this document are concerned only with policies that affect the entire distributed application program. Perhaps we will pursue multiple levels of policy in a later version of ArchOS.

There is another sense in which multiple levels of policy may be discussed -- a hierarchical sense. A facility at a given level of abstract may be decomposed into a set of mechanisms that are manipulated by means of a set of policy statements. Each of the mechanisms at that level of abstraction, in turn, are also decomposable at a lower level of abstraction into a set of lower-level mechanisms manipulated by a lower-level policy; and so on. At this time, it is difficult to see exactly how such a hierarchical decomposition of facilities will aid in their design or performance. For the present, we have decided to apply the notion of policy/mechanism separation at only a single, meaningful level.

(Our system was not intended to examine this policy/mechanism hierarchcy, and, once again, it is not central to the development of ArchOS as a research vehicle.)

Third, although we are currently employing only two client-defined policies, the primitives provided for the definition of policies by the application program are intended to be used for any other policies that may be defined. They are very generic in nature (associating a policy module with a special policy name and associating values with attribute names that may later be examined in carrying out policy decisions) and were intentionally selected to support a wide-range of policy definitions. We felt that a general approach to policy definition was more desirable than a specialized approach for each policy to be handled.

Finally, we have given the implementation of an application-defined process scheduling policy some thought. Since this facility must be accessed often and it must manipulate low level operating system objects (such as the queue of runnable processes), it would be advantageous for it to be resident in the kernel's memory space. We intend to have a method to allow at least that special case of an application-defined policy module be resident in kernel space. Of course, updating the process scheduling policy and coordinating the access of the policy module with client-space data will provide some additional complications, but these should not be too great.

4.4.3 ArchOS Primitives

The ArchOS primitives described in this report can be divided into two levels. The primitives which belong to the ArchOS kernel level are called *kernel* primitives and the primitives which are implemented above the kernel are called *system* primitives. This distinction will help an application designer but we did not provide any indication of which is which at this stage. The relation between the system and kernel primitives will be described in more detail in the System Architecture Specification document.

This section starts by briefly pointing out the important design problems for the complete set of ArchOS primitives. It then describes our design decisions for each primitive, reflecting the structure in the the previous chapter.

4.4.3.1 Important Design and Research Problems

In the design of the ArchOS primitives, we consider simplicity and uniformity to be the most important design goals. All of the ArchOS primitives should provide a uniform interface to a client. Thus, a primitive is accessed by a procedure (or function) invocation.

While we are trying to achieve simplicity, we also consider the primitive set's optimality as well as its completeness. Unfortunately, there is no formal notion of a complete set of primitives in a distributed operating system context [Tokuda 83a], so we have tried to evaluate the primitives' expressive power in various distributed applications.

- **Simplicity**:
  We have adopted a procedure (or function) call interface for all of our primitives. Even though the request message must be transferred to a suitable arobject by a *Request* primitive, it will be called by a procedure (or function) interface. It should be noted that our original intention was to reduce the number of primitives visible from a client process. For instance, a single *Create* primitive would be sufficient if the target language allowed "overloading" of procedures/functions. The current *CreateArobject* and *CreateProcess* could be united and used as follows:

  ```
  arobject-id = Create(arobj-name);
  process-id = Create(process-name);
  ```

  One weak point is the lack of a notion of default parameters. For example, the default value of the node-id argument in the *CreateProcess* primitive could be set to "ANY-NODE", so that a caller would not need to specify this optional argument every time. However, the current ArchOS interface requires that the client must specify all parameter values explicitly at calling time. (due to the limitation of C language.)

- **Uniformity**:
  We have provided a uniform syntax as well as uniform (i.e., network transparent or location independent) semantics for each primitive. For instance, communication primitives provide the identical semantics for local and remote communications. Arobject and process management primitives also provide uniform semantics for creating and destroying an instance.

- **Optimality and Completeness**:
  While attempting to minimize the number of primitives in ArchOS, we paid careful attention to assure that the full expressive power of the primitive set was maintained. On the other hand, many primitives were added to support new facilities such as transaction and policy management. Also due to some language constraints, we needed to increase the number of primitives a little, but the current primitive set can be used to construct an extremely diverse set of distributed applications.

4.4.3.2 Arobject/Process Management

We view an arobject as a basic module for embodying a distributed abstract data type. In particular, we decided to treat an arobject as an *active* system entity rather than a *passive* entity. There are many advantages and disadvantages related to this decision. First, we can easily define an autonomous module. It is easy for an arobject not only to initialize its computational state by itself, but also to recover its computational state by itself. In other words, by having a single INITIAL process in each arobject, a designer can give responsibility for recovery to the INITIAL process. Second, unlike

traditional procedure invocation in abstract data types, a caller cannot invoke an operation of an arobject in a *master-slave* manner, but must use a form of rendezvous. The receiver therefore has the right to accept, reject, or delay the requested function. Finally, the degree of parallelism within an arobject can be dynamically changed in many ways. For instance, a process can be created in a different node to perform a requested computation on demand or many processes can be pre-created to accept a particular type of request.

We provided a completely network transparent arobject/process management. That is, creation and destruction of arobjects and processes can be performed without knowing the location of the target arobject or process.

The following functionalities are not adopted for the current ArchOS:

- **Multiple, simultaneous creation of arobjects and processes**
  This might be useful to instantiate a replicated arobject simultaneously. However, it creates more conflict with other optional arguments such as node-id. Thus, it must be invoked once for each arobject or process instance in the current system.

- **Hierarchical dependency among arobjects**
  There are no hierarchical dependency among arobjects except for nested arobjects in a distributed program. Thus, a single arobject can be created and destroyed with no effect on the rest of the external the arobjects in the same distributed program. However, it should be noted that nested arobjects are killed when enclosing arobject is killed.

- **Inheritance relationship among arobjects**
  There are no inheritance relationship among arobjects. In modern programming languages, such as Smalltalk-80 [Goldburg 83], Flavor [Weinreb 81], and LOOPS [Bobrow 81], the inheritance relationships among objects have shown great advantages, improving the structural sharing among abstract objects and simplifying the modification of the existing objects. We considered a multiple inheritance relationship among arobjects, but there were many difficult problems in providing a way to forward a request message to its super(class) arobject. In our arobject paradigm, we wished to avoid an unnecessarily (deep) hierarchy among arobjects; thus we did not adopt the inheritance relationship. However, the current model can support a nested arobject which is private to the outer arobject.

- **Hierarchical dependency among cooperating processes**
  There is no hierarchical relationship among cooperating processes. For instance, in ADA [Ada 83], the termination of a task depends upon the termination of its inner blocks' tasks. Thus, the termination of a higher-level task may be delayed. In Shoshin [Tokuda 83b], every process must belong to a family tree which indicates such termination dependencies. A process in Shoshin can be attached to or detached from the creator's family tree at creation time. This dependency information might be useful during the debugging phase, but the current system does not support this facility. That is, a process can kill other processes or terminate itself without causing any additional killing among communicating processes.

process accesses a private data object from a remote node, a conventional remote procedure call will be used. If such a data object and the calling process are located in the same node, then the normal procedure call will be used with identical semantics to the RPC case.

- **Built-in timeout facility in communication primitives**
  In order to bound the execution time of communication activity, we considered adding one more argument, namely a timeout value, to each communication primitive. However, we preferred to give the user a timeout facility outside the communication primitives. It should be noted that each communication primitive may contain a compound transaction so that it is also bounded internally.

4.4.3.4 Synchronization

There are two levels of synchronization support in ArchOS. One is a critical region for controlling shared private data objects and the other is explicit locking primitives for controlling inter-transaction activities.

There are many problems with the critical region construct. For instance, if a process dies within a critical region, the state of the shared objects cannot easily be restored. It is also difficult to bound the total waiting time as well as the execution time of the critical region.

Despite these difficulties, a critical region scheme was adopted, since we expect that creating a compound transaction for such a shared object may be unnecessary in certain situations. For instance, there are cases in which permanency of data objects is sufficient without providing full failure atomicity.

The explicit lock and release primitives were necessary to control concurrent transaction activities. It is clear that by using a discrete lock type, we could encounter a deadlock. In case of a simple deadlock, ArchOS might detect and resolve it; however, in general, it will remain the user's responsibility to detect and resolve deadlock problems.

On the other hand, by using a tree lock protocol a user can avoid the deadlock problem in some cases. However, a user must declare the dependency among locks in a tree by using the *CreateLock* primitive. (See Section 4.4.2.7).

The current synchronization management does not support or adopt the following functionality:

- **Error recovery in a critical region scheme**
  We believe that any shared object which requires failure atomicity should be accessed from a transaction. For instance, we can use a compound transaction to replace the critical region by using the CT{ ... } construct.

- **Timeout (exception) handling within a critical region**
  The critical region construct simply takes a timeout parameter to bound the total execution time of the caller. This timeout mechanism does not provide a corresponding exception handler. Thus a user must provide its function.

4.4.3.5 Transaction/Recovery Management

A compound or elementary transaction construct creates a new transaction scope in a client process. Within this scope, a client can access atomic objects as if these computational steps were executed alone. However, there are several restrictions on these steps needed to maintain the basic properties of transactions. These restrictions are as follows:

- Between any two transactions, a transaction cannot pass any its computational state to the other transaction by using a non-atomic data object. The following Example 1 shows two illegal transaction scopes in a single process.

Example 1: Computational State Passing via a Non-atomic Data Object

```
Arobject Server body
{ ...
  process INITIAL(parameters)
  {     . . .
        NormalType State1, State2;    /* Non-Atomic objects */

        ET1(timeout){
            State1 = Compute1(arg1, ..., argk)
        }
        . . .
        ET2(timeout){
            State2 = Compute2(State1, arg1, ..., argk)
        }
  }
}
```

In Example 1, the illegal (state) passing occurs within the INIT process by using the non-atomic data object, $State_1$. Since ET_2 depends on the value of $State_1$, the transaction property of ET_2 will not maintained.

- From two independent transaction scopes, two arobjects cannot communicate with each other by using a communication primitive or a shared abstract data object. This is an example of so-called *cooperating transactions* [Sha 83b] and the current ArchOS model does not support it.

  For instance, the following two examples show normal and cooperating transaction scopes.

Example 2: A Normal Transaction Scope

```
Arobject Requestor body
{ ...
  process INITIAL(parameters)
  {     . . .
        ET(timeout){
            Request(Server-Aid, SERVICE, reqmsg, repmsg);
            . . . steps . . .
        }
  }
}

Arobject Server body
{ ...
  process INITIAL(parameters)
  {     . . .
        while (true) {
                t-opr-aid = AcceptAny(ANYOPR, reqmsg);

                CT(timeout){   /* begin compound transaction */
                    . . . steps . . .
                    do←service(t-opr-aid.opr, reqmsg, replymsg);
                }   /* end transaction */

                Reply(t-opr-aid.tid, replymsg);
        }
  }
}
```

Example 3: A Cooperating Transaction Scope

```
Arobject Requestor body
{ ....
  process INITIAL(message)
  {     . . .
        ET(timeout){
            Request(Server-Aid, SERVICE, reqmsg, repmsg);
            . . . steps . . .
        }
  }
}

Arobject Server body
{ ...
  process INITIAL(message)
  {     . . .
        while (true) {
                CT(timeout){  /* begin compound transaction */
                     t-opr-aid = AcceptAny(ANYOPR, reqmsg);
                     . . . steps . . .
                     do-service(t-opr-aid.opr, reqmsg, replymsg);
                     Reply(t-opr-aid.tid, replymsg);
                }    /* end transaction */
        }
  }
}
```

In Example 2, it is easy to determine the execution sequence of the two transactions. That is, ET in the requestor happened before CT in the server. On the other hand, Example 3 shows two concurrent arobjects, namely two INITIAL processes, communicating with each other from within two separate transactions.

- If an elementary transaction contains a nested compound transaction, there is a possibility of deadlock due to the nature of the lock management. The following example shows the possibility of deadlock within a single transaction.

Example-4: A Deadlock within a Single Transaction

```
Arobject Server body
{ ...
  Process Server(message)
  {     . . .
        ET(timeout) {  /* begin elementary transaction */
            . . . ET step 1 . . .
            sval = SetLock(DISCRETE, Lx, WRITE) /* get a lock on ob
            . . . operate on X . . .

            CT(timeout){  /* begin compound transaction */
                . . . CT steps . . .
                /* get a lock on object X */
                sval = SetLock(DISCRETE, Lx, WRITE)
                <<. . . operate on X . . .>>  /* deadlock! */
            }   /* end transaction */
            . . . ET step 3 . . .
        }   /* end transaction */
}
```

The problem occurs when the nested compound transaction tries to obtain a lock on object X. Since the lock was already taken by the top level transaction ET, this compound transaction will abort due to timeout.

The current transaction/recovery management facility does not support or adopt the following functionality:

- **Cooperating transactions**
  Cooperating transactions are not supported as explained above.

- **Detection of deadlock in the lock management**
  Complete detection in the locking facility is not provided. A detection mechanism, in general, was left for the application designer. However, the proper use of tree-locks may help the client to avoid deadlock.

- **Automatic generation of a compensate action for a compound transaction**
  It is very complicated to automatically generate a compensate action for each operation involved by a compound transaction. Thus, this was left for the application designer. ArchOS supports only the automatic execution of well-defined compensate actions when a compound transaction is aborted.

- **Automatic lock management for a shared object**
  To determine the proper set of locks and their locking rules from the program is not an easy task, and often the amount of sharing obtained is limited due to simple-minded locking rules. The lock management is also left for the application designer.

4.4.3.6 File Management

The current file management in ArchOS supports a single-level file structure and does not provide any directory structure for users. However, the system can provide at least three kinds of file properties. First, a *normal* file has the data portion of the file arobject in volatile storage. Second, a *permanent* file's data portion is allocated in permanent (non-volatile) storage. Finally, an *atomic* file has the data portion which is declared as an atomic data object.

Since there is no notion of file protection at this level, a protection domain can be build based on the scope of the reference name.

The current file management does not support or adopt the following functionality. (Note that the following features are not permanently removed from ArchOS. These functionalities may be easily adopted on top of the current file arobject's structure).

- **Directory structure**
  There is no directory structure in the current file system. A client must use a flat file name space to classify files.

- **A replicated atomic file type**
  This replicated file type must vary in terms of its access protocols and replication schemes. Thus, there is no system supported replicated file type so far.

- **File sharing and protection scheme**
  There is no conventional access list for a file object, rather each file will have a set of locks to control concurrent file accesses.

It should be noted that if a normal or permanent file is accessed from a transaction scope, all of the transaction properties will not be provided. The file must be an atomic file type in order to fully utilize the transaction facility.

4.4.3.7 I/O Device Management

The I/O device management provides a set of access primitives for normal or special devices in the system. These primitives are similar to the ones which are used for file management, but ArchOS does not provide complete compatibility between the two.

The current I/O device management does not support or adopt the following functionality.

- **Full compatibility between devices and files**
  The main reason was that it is difficult to cover all different, specialized devices as standard (i.e., atomic, permanent or normal) files.

- **Transaction facilities are not supported on the *real* devices**

Unlike arobjects, a user cannot create a transaction for a sequence of device operations. This is because "undo"s are often impossible following real actions taken on these devices.

4.4.3.8 Time Management

The time management primitives provide functions for obtaining getting time information and/or setting/resetting scheduling parameters for performing time-driven scheduling. This time-driven scheduling is based on ArchOS' best effort scheduling model. Since this model requires at least *request*, *delay*, *deadline*, and *estimated execution times* as basic parameters, the *Delay* and *Alarm* primitives were designed to provide these parameters from a client process.

There are several approaches to the provision of these functions in a real time operating system, but these were chosen in an attempt to provide maintainability and modularity as well as time control. For example, many systems provide a process time-out setting to be made at which an application process will be scheduled. This frequently requires breaking a module at the delay point into two processes, rendering the processing difficult to read and understand. The *Delay* and *Alarm* primitives provide for such delays to be coded inline in the application without breaking the process into multiple processes.

At the same time, the deadline-driven scheduling algorithm to be used in ArchOS is provided with the time parameters needed to apply deadline policies. A part of the research being conducted on the Archons project concerns process scheduling in the presence of application-defined deadlines. It is intended that application-defined policies will be used to control deadline-driven scheduling which will attempt to meet deadlines or minimize the damage if insufficient resources are available to meed deadlines.

The current time management does not support or adopt the following functionality:

- **Asynchronous Delay or Alarm primitive**
  Since we would like to avoid interrupt-driven timeout routines, the Delay and Alarm primitives are blocking primitives. This reduces the complexity required in the structure of a process body. To provide this type of asynchronous handling, the user may create a new (time) process to avoid the blocking.

4.4.3.9 Policy Management

The policy management primitives provide functions which implement the basic structure of our policy module. The policy set arobject maintains a set of policy modules in a distributed program. A policy module represents a real policy and maintained by by ArchOS or the policy set arobject.

The current policy management does not support or adopt the following functionality:

- **Dynamic addition/deletion of a new user-defined policy module**
  The current model can only support predefined set of user policies, thus a new user-defined policy cannot be recognized by ArchOS.

4.4.3.10 System Monitoring and Debugging Support

The system monitoring and debugging facilities provide various abilities to control the behavior of cooperating arobjects and their processes during execution. All of the primitives for system monitoring and debugging can only be used by a specially privileged arobject, so that a normal client arobject cannot issue any of them.

The current system monitoring and debugging facilities do not support or adopt the following functionality:

- **Single-step function for tracing process activity**
  A single step function is not supported yet, but may be added in the future.

- **Creating an arbitrary break point in an arobject**
  A client cannot set an arbitrary break point in an arobject. The available breaking points are only the communication points where a process sends or receives a message. A monitoring process may trap the following execution step by using the CaptureComm or WatchComm primitives.

- **Specialized debug functions such as *Redo* or *Undo* operation for arbitrary functions**
  These facilities should be built by using the current facilities, rather than creating a set of new primitives.

A. Program Examples

This appendix presents an example problem involving a distributed data object along with two potential solutions employing arobjects and the ArchOS system primitives.

A.1 The Problem: A Distributed Directory with Partially Replicated Data

In this example, the problem is to implement a distributed directory, where the directory data is physically spread among several processing nodes with each directory entry replicated at multiple nodes in order to improve reliability. Notice that there is no guarantee that every node has a complete copy of the directory information.

In particular, the problem is to construct a distributed directory of some fixed length, where each entry in the directory associates a name (a string of characters) with a value (in the examples, the values used are assumed to be integers). The data in the directory must be partially replicated at several different processing nodes, and directory users must always have a consistent view of the directory.

This type of distributed directory represents a class of data object that will be common in an ArchOS client's distributed program---an object that is highly robust (due to the presence of multiple copies of each critical data item at physically separate nodes) but does not require that the user of this data object be aware of the physical distribution involved in the implementation. (In fact, this implementation is invoked by the user in exactly the same manner as a centralized data object.)

A.2 The General Approach to the Problem

The solutions presented in the next two sections are both built upon the same fundamental approach. The directory data is contained in several different arobjects (called *directory-copy* arobjects), located at different processing nodes. These arobjects each contain some portion of the total directory state, but it is unlikely that any two of them contain exactly the same data or that any one of them contains all of the current directory data.

Another entity is provided to consistently access and maintain the directory-copy arobjects so that the directory user "sees" a single directory object (called the *directory* arobject). This entity takes a different form in the two solutions that follow, and this is their major distinguishing feature. In the first example, one or more *directory* arobjects accept directory operation invocations and then service

them by making the appropriate invocations on the directory-copy arobjects. All of the arobjects in this solution are separate entities, and so this solution views the organization of arobjects as being quite "flat."

In the second solution, the entire directory service is provided by a single arobject. This arobject is physically distributed over several nodes and has placed multiple processes throughout the system to accept directory operation invocations. In addition, the directory-copy arobjects are embedded within the directory arobject as private arobjects in the second solution. As a result, this solution views arobjects as being organized in a hierarchical manner.

ArchOS supports both views of the organization of arobjects (flat or hierarchical). Neither of them appears to be better than the other for this example, but the client is free to organize arobjects in the manner that seems most advantageous for the application at hand.

Both solutions manage the coordination of the multiple directory copies in the same way: by applying the notion of using only a quorum [Gifford 79, Herlihy 84] of directory copies in order to carry out any operation on the distributed directory. Using this method allows the directory to continue operating smoothly even if several directory copies are unavailable at any given time. Each directory operation (lookup, add, delete) requires only a quorum of directory copies be involved to perform that operation. In order to determine the latest value of a given name, the directory-copy arobjects also maintain a version number with each entry name. (The current value of a given name corresponds to the entry for that name with the highest version number in any directory copy.)

Also, notice that the transaction facility has been used in order to guarantee the consistency of the directory information at all times. Elementary transactions have been used for the most part since the directory operations invoked by a user of the directory arobject may be part of a larger transaction, and in that case, if compound transactions had been used, the consistency of the directory could no longer be guaranteed.

A.3 Solution 1: Two Cooperating Classes of Arobjects

As outlined in the previous section, this solution uses several separate directory and directory-copy arobjects distributed throughout the system. The code for these arobject types is assumed to be contained in two files: "directory.arb" contains the code for the directory arobject and "dir-copy.arb" contains the code for the directory-copy arobject.

```
/*****************************************************************
*                                                                *
*   This code is contained in the file "directory.arb"           *
*                                                                *
*****************************************************************/

#define DIR-COPIES              2*N+1
#define READ-QUORUM             N+1
#define WRITE-QUORUM            N+1
#define DEL-QUORUM              2*N+1
/*  If a DEL-QUORUM cannot be formed, NULL can be written to the value
        field of a new version of thedesired name.  Later, when all
        copies are available, a delete can again be attempted.  */

arobject directory specification
{
    char *name;
    int value;
    STATUS result;

    operation add(name, value) --> (result);
    operation delete(name) --> (result);
    operation find(name) --> (result, value);
}

arobject directory body
{
    ***** insert message type declarations here *****

  /*  Private Abstract Data Type Definitions  */
  private-abstract-data-type DCNAME = {
    permanent AROBJ-REFNAME dir-copy-name;

  /*  Procedures to Operate on Atomic Data Items  */

    procedure store(dcname)
    AROBJ-REFNAME dcname;
    {
        dir-copy-name = dcname;
    }

    AROBJ-REFNAME function get()
    {
        return(dir-copy-name);
    }
  }     /*  end of private-abstract-data-type DCNAME  */

  DCNAME dcname;

    process INITIAL(dircopyname)
    AROBJ-REFNAME dcname;
    {
        MESSAGE *requestmsg;
        OPERATION opr;
```

```
    dcname.store(dircopyname);

    while(TRUE) {
        AcceptAny(ANYOPR, requestmsg);
        opr = msg-header-operation(requestmsg);
        CreateProcess(opr, requestmsg);
    }
}

process add(add-requestmsg)
{
    char *name;
    int value;
    int i;
    STATUS rd-result;
    int val[READ-QUORUM];
    int version[READ-QUORUM];

    TIME timeout = TIMEOUT;
    TRANSACTION-ID tid, trans-id;

    PID pid;
    struct add-replymsg {
        STATUS result;
    }

    strcpy(name, requestmsg.body.name);
    value = requestmsg.body.value;

    pid = msg-header-caller(add-requestmsg);
    tid = msg-header-tid(add-requestmsg);

    ET(timeout) {
        trans-id = SelfTid();

        read-quorum(name, val, version, rd-result);
        if (rd-result != OK) {
            Reply(pid, tid, {FAIL});
            AbortTransaction(trans-id);
        }
        max-version = -1;
        for (i=1; i<= READ-QUORUM; i++)
            max-version = max(max-version, version[i]);
        write-quorum(name, value, max-version+1, wr-result);
        add-replymsg.body.result = wr-result;
        Reply(pid, tid, add-replymsg);
    }
    if (IsAborted(trans-id)) {
            /* handle error condition here */
    }
}

procedure read-quorum(name, value, version, result)
char *name;
int value[READ-QUORUM];
```

```
    int version[READ-QUORUM];
    STATUS result;
    {
        int count;

        TRANSACTION-ID tid;
        FIND-REPLYMSG find-replymsg;

        count = 0;
        /*  initiate timeout  */
        tid = RequestAll(dcname.get(), find, {name});
        while (count < READ-QUORUM) {
            GetReply(tid, find-replymsg);
            if (find-replymsg.body.result == OK) {
                count++;
                value[count] = find-replymsg.body.value;
                version[count] = find-replymsg.body.version;
            }
        }
        /*  Need to fix case where quorum cannot be established.  */
        AbortIncompleteTransaction(tid);
        result = OK;
    }

    procedure write-quorum(name, value, version, result)
    char *name;
    int value, version;
    STATUS result;
    {
        int count;

        TRANSACTION-ID tid;
        struct add-roplymsg ...

        count = 0;
        /*  initiate timeout  */
        tid = RequestAll(dcname.get(), add, {name, value, version})
        if (add-replymsg.body.result == OK) count++;
        while (count < WRITE-QUORUM) {
            GetReply(tid, add-replymsg);
            if (add-replymsg.body.result == OK) count++;
        }
        /*  Need to fix case where quorum cannot be found.  */
        AbortIncompleteTransaction(tid);
        result = OK;
    }
}       /*  End of directory arobject body  */
```

```
/*****************************************************************
*                                                                *
*   This code is contained in the file "dir-copy.arb"            *
*                                                                *
*****************************************************************/

arobject directory-copy specification
{
    char name[MAXSIZE];
    int value;
    int version-no;
    status result;

    operation add(name, value, version-no) --> (result);
    operation delete(name) --> (result);
    operation find(name) --> (result, value, version-no);
}

arobject directory-copy body
{
  /*  Private Abstract Data Type Definitions  */
  private-abstract-data-type TABLE = {
    TREE-LOCK-ID t-lock[TABLESIZE];
        /* define tree lock structure as a "linear" tree, such that
                t-lock[1] is the root and has child t-lock[2];
                t-lock[2] has child t-lock[3];
                and so on. */
    atomic struct table[TABLESIZE] {
        char *name;
        int value;
        int version;
    }

  /*  Procedures to Operate on Atomic Data Items  */

    procedure init-table()
    {
        int i;
        TIME timeout = TIMEOUT;
        TRANSACTION-ID tid;

        /*  define tree lock structure  */
        t-lock[1] = CreateLock(NULL-LOCK-ID);
        for (i=2; i<=TABLESIZE; i++) {
            t-lock[i] = CreateLock(t-lock[i-1]);
        }

        CT(timeout) {
            tid = SelfTid();
            for (i=1; i<=TABLESIZE; i++) {
                SetLock(TREE-LOCK, t-lock[i], WRITE);
                table[i].name = NULL;
                ReleaseLock(TREE-LOCK, t-lock[i], WRITE);
            }
        }
```

```
    if (IsAborted(tid)) {
        /* handle error condition */
    }
}

procedure lookup(name, index)
char *name;
{
    int index, i;

    TIME timeout = TIMEOUT;

    index = -1;
    ET(timeout) {
        for (i=1; i<=TABLESIZE; i++) {
            SetLock(TREE-LOCK, t-lock[i], READ);
            if (strcmp(table[i].name, name) == NULL) {
                index = i;
                breakfor;
            }
            ReleaseLock(TREE-LOCK, t-lock[i], READ);
        }
    }
}

STATUS function enter(index, name, value, version-no)
char *name;
int index, value, version-no;
{
    TIME timeout = TIMEOUT;
    TRANSACTION-ID tid;

    ET(timeout) {
        tid = SelfTid();

        SetLock(TREE-LOCK, t-lock[index], WRITE);
        if (table[index].name != NULL || table[index].name != name)
            AbortTransaction(tid);
        strcpy(table[index].name, name);
        table[index].value = value;
        table[index].version = version-no;
    }
    if (IsCommitted(tid)) return(OK);
    else return(FAIL);
}

procedure get-fields(index, name, value, version-no, result)
char *name;
int index, value, version-no;
STATUS result = FAIL;
{
    TIME timeout = TIMEOUT;
    TRANSACTION-ID tid;

    ET(timeout) {
```

```
            tid = SelfTid();

            SetLock(TREE-LOCK, t-lock[index], READ);
            strcpy(name, table[index].name);
            value = table[index].value;
            version-no = table[index].version;
        }
        if (IsCommitted(tid)) result = OK;
    }
}   /*  end of private-abstract-data-type TABLE    */

TABLE tab;

  process INITIAL()
  {
      OPERATION opr;

      tab.init-table();

      while (TRUE) {
          AcceptAny(ANYOPR, requestmsg);
          opr = msg-header-operation(requestmsg);
          CreateProcess(opr, requestmsg);
      }
  }

  process add(add-requestmsg)
  ADD-REQUESTMSG add-requestmsg;
  {
      char *name;
      int value, version-no;
      int index;
      STATUS stat, enter();
      TIME timeout = TIMEOUT;

      struct add-replymsg {
          STATUS result = FAIL;
      }

      strcpy(name, add-requestmsg.body.name);
      value = add-requestmsg.body.value;
      version-no = add-requestmsg.body.version;

      ET(timeout) {
          index = lookup(name);
          if (index < 0) index = lookup(NULL);
          if (index < 0) AbortTransaction(SelfTid());
          stat = enter(index, name, value, version-no);
          if (stat != OK) AbortTransaction(SelfTid());
          add-replymsg.body.result = OK;
      }
      Reply(msg-header-caller(add-requestmsg),
              msg-header-tid(add-requestmsg), add-replymsg);
  }
```

```
    process find(find-requestmsg)
    FIND-REQUESTMSG find-requestmsg;
    {
        char *name;
        int index, value, version-no;
        char *entry-name;
        STATUS stat;

        struct find-replymsg ...;
        PID pid;
        TRANSACTION-ID tid;

        strcpy(name, find-requestmsg.body.name);
        pid = msg-header-caller(find-requestmsg);
        tid = msg-header-tid(find-requestmsg);

        index = lookup(name);
        if (index < 0) Reply(pid, tid, {NOT-FOUND});
        else {
            get-fields(index, entry-name, value, version-no, stat);
            if (strcmp(name, entry-name) == NULL)
                Reply(pid, tid, {OK, value, version-no});
            else Reply(pid, tid, {FAIL});
        }
    }
}       /*  End of directory-copy arobject body  */
```

A.4 Solution 2: A Single, Distributed Arobject

As previously discussed, this solution implements the entire directory server as a single arobject in which several directory-copy arobjects have been included as private arobjects. As in the previous solution, the code for the directory arobject is located in "directory.arb," while the code for the directory-copy arobject is located in "dir-copy." In fact, the code for the directory-copy arobject is exactly the same as in the first solution; however, the code for the directory arobject has been changed somewhat. The greatest change has taken place in the INITIAL process of the directory arobject; also, the directory arobject now contains a statement that identifies the description of the directory-copy arobject as a private arobject.

```
/*****************************************************************
*                                                                *
*   This code is contained in the file "directory.arb"           *
*                                                                *
*****************************************************************/

#define DIR-COPIES              2*N+1
#define READ-QUORUM             N+1
#define WRITE-QUORUM            N+1
#define DEL-QUORUM              2*N+1
/*  If a DEL-QUORUM cannot be formed, NULL can be written to the value
        field of a new version of thedesired name.  Later, when all
        copies are available, a delete can again be attempted.  */

arobject directory specification
{
    char *name;
    int value;
    STATUS result;

    operation add(name, value) --> (result);
    operation delete(name) --> (result);
    operation find(name) --> (result, value);
}

arobject directory body
{
    ***** insert message type declarations here *****

  /*  Private Abstract Data Type Definitions  */
  private-abstract-data-type DCNAME = {
    permanent AROBJ-REFNAME dir-copy-name;

  /*  Procedures to Operate on Atomic Data Items  */

    procedure store(dcname)
    AROBJ-REFNAME dcname;
    {
        dir-copy-name = dcname;
    }

    AROBJ-REFNAME function get()
    {
        return(dir-copy-name);
    }
  }     /*  end of private-abstract-data-type DCNAME  */

  DCNAME dcname;

    private-arobject directory-copy = "dir-copy.arb";

    process INITIAL()
    {
        NODE node[TOTAL-NODES] = {NODE1, NODE2, ... ,NODEN};
        int i;
```

```
    AID aid;

    for (i=1; i<=COPIES; i++) {
        aid = CreateArobject(directory-copy, NULL, node[i]);
        BindArobjectName(aid, "dir-copy");
    }
    dcname.store("dir-copy");
    for (i=1; i<=SERVERS; i++)
        CreateProcess(accept-invocs, NULL, node[i]);
}

process accept-invocs()
{
    MESSAGE *requestmsg;
    OPERATION opr;

    while(TRUE) {
        AcceptAny(ANYOPR, requestmsg);
        opr = msg-header-operation(requestmsg);
        CreateProcess(opr, requestmsg);
    }
}

process add(add-requestmsg)
{
    char *name;
    int value;
    int i;
    STATUS rd-result;
    int val[READ-QUORUM];
    int version[READ-QUORUM];

    TIME timeout = TIMEOUT;
    TRANSACTION-ID tid, trans-id;

    PID pid;
    struct add-replymsg {
        STATUS result;
    }

    strcpy(name, requestmsg.body.name);
    value = requestmsg.body.value;

    pid = msg-header-caller(add-requestmsg);
    tid = msg-header-tid(add-requestmsg);

    ET(timeout) {
        trans-id = SelfTid();

        read-quorum(name, val, version, rd-result);
        if (rd-result != OK) {
            Reply(pid, tid, {FAIL});
            AbortTransaction(trans-id);
        }
        max-version = -1;
```

```
        for (i=1; i<= READ-QUORUM; i++)
            max-version = max(max-version, version[i]);
        write-quorum(name, value, max-version+1, wr-result);
        add-replymsg.body.result = wr-result;
        Reply(pid, tid, add-replymsg);
    }
    if (IsAborted(trans-id)) {
        /* handle error condition here */
    }
}

procedure read-quorum(name, value, version, result)
char *name;
int value[READ-QUORUM];
int version[READ-QUORUM];
STATUS result;
{
    int count;

    TRANSACTION-ID tid;
    FIND-REPLYMSG find-replymsg;

    count = 0;
    /*  initiate timeout  */
    tid = RequestAll(dcname.get(), find, {name});
    while (count < READ-QUORUM) {
        GetReply(tid, find-replymsg);
        if (find-replymsg.body.result == OK) {
            count++;
            value[count] = find-replymsg.body.value;
            version[count] = find-replymsg.body.version;
        }
    }
    /*  Need to fix case where quorum cannot be established.  */
    AbortIncompleteTransaction(tid);
    result = OK;
}

procedure write-quorum(name, value, version, result)
char *name;
int value, version;
STATUS result;
{
    int count;

    TRANSACTION-ID tid;
    struct add-replymsg ...

    count = 0;
    /*  initiate timeout  */
    tid = RequestAll(dcname.get(), add, {name, value, version})
    if (add-replymsg.body.result == OK) count++;
    while (count < WRITE-QUORUM) {
        GetReply(tid, add-replymsg);
        if (add-replymsg.body.result == OK) count++;
```

```
        }
        /*  Need to fix case where quorum cannot be found.  */
        AbortIncompleteTransaction(tid);
        result = OK;
    }
}       /*  End of directory arobject body  */
```

```
/*****************************************************************
*                                                                *
*   This code is contained in the file "dir-copy.arb"            *
*                                                                *
*****************************************************************/

arobject directory-copy specification
{
    char name[MAXSIZE];
    int value;
    int version-no;
    status result;

    operation add(name, value, version-no) --> (result);
    operation delete(name) --> (result);
    operation find(name) --> (result, value, version-no);
}

arobject directory-copy body
{
  /*  Private Abstract Data Type Definitions  */
  private-abstract-data-type TABLE = {
    TREE-LOCK-ID t-lock[TABLESIZE];
        /* define tree lock structure as a "linear" tree, such that
                t-lock[1] is the root and has child t-lock[2];
                t-lock[2] has child t-lock[3];
                and so on. */
    atomic struct table[TABLESIZE] {
        char *name;
        int value;
        int version;
    }

  /*  Procedures to Operate on Atomic Data Items  */

    procedure init-table()
    {
        int i;
        TIME timeout = TIMEOUT;
        TRANSACTION-ID tid;

        /*  define tree lock structure  */
        t-lock[1] = CreateLock(NULL-LOCK-ID);
        for (i=2; i<=TABLESIZE; i++) {
            t-lock[i] = CreateLock(t-lock[i-1]);
        }

        CT(timeout) {
            tid = SelfTid();
            for (i=1; i<=TABLESIZE; i++) {
                SetLock(TREE-LOCK, t-lock[i], WRITE);
                table[i].name = NULL;
                ReleaseLock(TREE-LOCK, t-lock[i], WRITE);
            }
        }
```

```
    if (IsAborted(tid)) {
        /* handle error condition */
    }
}

procedure lookup(name, index)
char *name;
{
    int index, i;

    TIME timeout = TIMEOUT;

    index = -1;
    ET(timeout) {
        for (i=1; i<=TABLESIZE; i++) {
            SetLock(TREE-LOCK, t-lock[i], READ);
            if (strcmp(table[i].name, name) == NULL) {
                index = i;
                breakfor;
            }
            ReleaseLock(TREE-LOCK, t-lock[i], READ);
        }
    }
}

STATUS function enter(index, name, value, version-no)
char *name;
int index, value, version-no;
{
    TIME timeout = TIMEOUT;
    TRANSACTION-ID tid;

    ET(timeout) {
        tid = SelfTid();

        SetLock(TREE-LOCK, t-lock[index], WRITE);
        if (table[index].name != NULL || table[index].name != name)
            AbortTransaction(tid);
        strcpy(table[index].name, name);
        table[index].value = value;
        table[index].version = version-no;
    }
    if (IsCommitted(tid)) return(OK);
    else return(FAIL);
}

procedure get-fields(index, name, value, version-no, result)
char *name;
int index, value, version-no;
STATUS result = FAIL;
{
    TIME timeout = TIMEOUT;
    TRANSACTION-ID tid;

    ET(timeout) {
```

```
            tid = SelfTid();

            SetLock(TREE-LOCK, t-lock[index], READ);
            strcpy(name, table[index].name);
            value = table[index].value;
            version-no = table[index].version;
        }
        if (IsCommitted(tid)) result = OK;
    }
}   /*  end of private-abstract-data-type TABLE    */

TABLE tab;

  process INITIAL()
  {
      OPERATION opr;

      tab.init-table();

      while (TRUE) {
          AcceptAny(ANYOPR, requestmsg);
          opr = msg-header-operation(requestmsg);
          CreateProcess(opr, requestmsg);
      }
  }

  process add(add-requestmsg)
  ADD-REQUESTMSG add-requestmsg;
  {
      char *name;
      int value, version-no;
      int index;
      STATUS stat, enter();
      TIME timeout = TIMEOUT;

      struct add-replymsg {
          STATUS result = FAIL;
      }

      strcpy(name, add-requestmsg.body.name);
      value = add-requestmsg.body.value;
      version-no = add-requestmsg.body.version;

      ET(timeout) {
          index = lookup(name);
          if (index < 0) index = lookup(NULL);
          if (index < 0) AbortTransaction(SelfTid());
          stat = enter(index, name, value, version-no);
          if (stat != OK) AbortTransaction(SelfTid());
          add-replymsg.body.result = OK;
      }
      Reply(msg-header-caller(add-requestmsg),
              msg-header-tid(add-requestmsg), add-replymsg);
  }
```

```
    process find(find-requestmsg)
    FIND-REQUESTMSG find-requestmsg;
    {
        char *name;
        int index, value, version-no;
        char *entry-name;
        STATUS stat;

        struct find-replymsg ...;
        PID pid;
        TRANSACTION-ID tid;

        strcpy(name, find-requestmsg.body.name);
        pid = msg-header-caller(find-requestmsg);
        tid = msg-header-tid(find-requestmsg);

        index = lookup(name);
        if (index < 0) Reply(pid, tid, {NOT-FOUND});
        else {
            get-fields(index, entry-name, value, version-no, stat);
            if (strcmp(name, entry-name) == NULL)
                Reply(pid, tid, {OK, value, version-no});
            else Reply(pid, tid, {FAIL});
        }
    }
}       /*  End of directory-copy arobject body  */
```

References

[Ada 83] *Ada Programming Language*
1983.

[Ahuja 82] Ahuja, Vijay.
Routing.
In *Design and Analysis of Computer Communication Networks*, chapter 7, pages 233-260. McGraw-Hill, 1982.

[Almes 83] Almes, G. T.; Black, A. P.; Lazowska, E. D.; Noe, J. D.
The Eden System: A Technical Review.
Technical Report 83-10-05, University of Washington, October, 1983.

[Bobrow 81] Bobrow, D. G.; Stefik, M.
The LOOPS Manual.
Technical Report KB-VLSI-81-13, Xerox Palo Alto Research Center, August, 1981.

[DeGroot 74] DeGroot, M.
Reaching a Consensus.
Journal of The American Statistical Association , March, 1974.

[Erman 73] Erman, L. D., Fennell, R. D., Lesser, V. R., and Reddy, D. R.
System Organizations for Speech Understanding.
Proceedings, Third International Joint Conference on Artificial Intelligence , August, 1973.

[Erman 79] Erman, L. D., and V. R. Lesser.
The Hearsay-II System: A Tutorial.
In W. A. Lea (editor), *Trends in Speech Recognition*, chapter 16. Prentice-Hall, 1979.

[Gifford 79] Gifford, D. K.
Weighted Voting for Replicated Data.
Operating Systems Review 13(5), December, 1979.

[Goldburg 83] Goldburg, A.; Robson, D.
Smalltalk-80: The Language and Its Implementation.
Addison-Wesley, 1983.

[Gray 77] James Gray.
Notes on Data Base Operating Systems.
Technical Report, IBM Research Laboratory, San Jose, 1977.

[Herlihy 84] Herlihy, M. P.
Replication Methods for Abstract Data Types.
PhD thesis, MIT Laboratory for Computer Science, 1984.

[Jensen 81a] Jensen, E. Douglas.
Decentralized Control.
In *Distributed Systems - Architecture and Implementation: An Advanced Course*, chapter 8. Springer-Verlag, 1981.

[Jensen 81b] Jensen, E. Douglas.
Decentralized Resource Management.
July, 1981.
Course Notes, C.E.A./I.N.R.I.A/E.D.F Ecole d'ete d'Informatique, Breau-sans-Nappe, France,

[Jensen 84] Jensen, E. D. and Pleszkoch, N.
ArchOS: A Physically Dispersed Operating System -- An Overview of its Objectives and Approach.
IEEE Distributed Processing Technical Committee Newsletter, Special Issue on Distributed Operating Systems , June, 1984.

[Kernighan 78] Brian W. Kernighan, Dennis M. Ritchie.
The C Programming Language.
Prentice-Hall, 1978.

[Lazowska 81] Lazowska, E. D., Levy, H. M., Almes, G. T., Fischer, M. J., Fowler, R. J., and Vestal, S. C.
The Architecture of the Eden System.
Operating Systems Review 15(5):148-159, December, 1981.

[Lynch 83] Lynch, Nancy A.
Concurrency Control for Resilient Nested Transactions.
In *Proceedings, Second SIGACT-SIGMOD Conference on the Principles of Database Systems.* ACM, 1983.

[Marschak 72] Marschak, J., and Radner, R.,
Economic Theory of Teams.
Yale University Press, 1972.

[Moss 81] Moss, J. E. B.
Nested Transactions: An Approach to Reliable Distributed Computing.
PhD thesis, Massachusetts Institute of Technology, April, 1981.

[Rashid 81] Rashid, Richard F. and George G. Roberston.
Accent: A Communication Oriented Network Operating System Kernel.
Technical Report CMU-CS-81-123, Department of Computer Science, Carnegie-Mellon University, April, 1981.

[Schwarz 82] Schwarz, Peter M. and Alfred Z. Spector.
Synchronizing Shared Abstract Types.
Technical Report CMU-CS-82-128, Department of Computer Science, Carnegie-Mellon University, 1982.

[Sha 83a] Sha, Lui, E. Douglas Jensen, Richard F. Rashid, and J. Duane Northcutt.
Distributed Cooperating Processes and Transactions.
In *Synchronization, Control, and Communication in Distributed Computing Systems.* Academic Press, 1983.

[Sha 83b] Sha, L., Jensen E. D, Rashid, R. F., and Northcutt, J. D.
Distributed Co-operating Processes and Trancactions.
Proceedings of ACM SIGCOMM symposium , 1983.

[Sha 84] Sha, Lui.
Synchronization in Distributed Operating Systems.
PhD thesis, Department of Electrical Engineering, Carnegie-Mellon University, 1984.
in preparation.

[Silberschatz 80] Silberschatz, A., and Z. Kedem.
Consistency in Hierarchical Database Systems.
Journal of the Association of Computing Machinery 27(1), January, 1980.

[Spice 79] Department of Computer Science.
Proposal for A Joint Effort in Personal Computing.
Technical Report, Carnegie-Mellon University, 1979.

[Tokuda 83a] Tokuda, H; Manning, E. G.
An Interprocess Communication Model for a Distributed Software Testbed.
Proceedings of ACM SIGCOMM symposium , 1983.

[Tokuda 83b] Tokuda, H., Radia, S. R., and Manning, E. G.
Shoshin OS: a Message-based Distributed Operating System for a Distributed Software Testbed.
Proceedings of the 16th Hawaii Int. Conf. on System Sciences , 1983.

[Weinreb 81] Weinreb, D.; Moon, D.
Lisp Machine Manual.
Technical Report, MIT Artificial Intelligence Laboratory, 1981.

[Wulf 81] Wulf, W. A.; Levin, R.; Harbison, S. P.
HYDRA/C.mmp: An Experimental Computer System.
McGraw-Hill, 1981.

3. ArchOS System/Subsystem Description

1. Introduction

This document constitutes the System/Subsystem Description of ArchOS (CDRL Item No. A003) for RADC Contract No. F30602-84-C-0063, Reconfigurable C^2 DDP System. This report describes the Archons operating system (ArchOS) at the system/subsystem level.

ArchOS is the operating system being designed and constructed as part of the Archons research project [Jensen 84]. The goal of the project is to conduct research into the issues of decentralization in distributed computing systems at the operating system level and below. The primary research issues being investigated within Archons are decentralized control, team and consensus decision making, transaction management, probabilistic algorithms, and architectural support in a real-time environment.

It is planned that ArchOS will be developed in three stages. The first, called the interim testbed version, is now underway and will handle a small set of Sun Workstations[1] interconnected by an Ethernet. It is expected that this operating system will constitute an existence proof of many of the basic concepts involved in our research as applied to the construction of a decentralized operating system. This system will later be followed by a more complete testbed operating system incorporating the lessons learned in the interim testbed construction. Finally, an analysis of the ArchOS structure is expected to result in the construction of specialized hardware for the purpose of executing a complete ArchOS system, and ArchOS will be rewritten to run on that hardware at that time.

This document consists of two reports, namely the ArchOS System Architecture Specification and the System Design Specification. Chapter 2 contains the description of the system architecture of ArchOS. The system architecture description summarizes the structure of ArchOS that supports the ArchOS client interface. ArchOS consists of the ArchOS kernel, subsystems, and system arobjects. The ArchOS kernel provides basic mechanisms for time-driven resource management. The subsystems together with system arobjects realize ArchOS facilities.

Chapter 3 contains the system design description of ArchOS. The system design description describes the structure and functionality of major components of ArchOS, such as the ArchOS kernel, the arobject/process management subsystem, the communication subsystem, the transaction subsystem, the file subsystem, the I/O device subsystem, the time-driven scheduler subsystem, and the monitoring and debugging subsystems.

---

[1] Sun Workstation is a trademark of SUN Microsystems, Inc.

1.1 Overview of ArchOS Development

In the previous RADC contract F30602-81-C-0297, we studied basic problems of decentralized resource management [Jensen 83a]. In particular, we created a new type of transaction, called a *compound transaction*, for operating system functions and developed its theory. We also studied policy/mechanism separation issues in Interprocess Communication and developed a decentralized algorithm testing environment, called *DATE*, supported by the 4.1 BSD UNIX operating system[2]. Furthermore, we explored issues in decentralized computer architecture focusing on processor architecture topics such as the separation of the operating system (OS) and application processors in a single node and the architecture of the OS processor. Finally, the first interim testbed facility was built based on a set of Sun workstations interconnected by an Ethernet.

Based on the results from these research efforts, we started the design and development of ArchOS at the end of 1983. During 1984, we reached several important milestones in our ArchOS development plan which were previously reported [Jensen 85]. First, we finalized our development plan and produced research requirements for ArchOS. The ArchOS development plan was described in our proposal [Jensen 83b], and the research requirements document was included in Chapter 2 of "Functional Description of ArchOS" [Jensen 85]. Second, we developed the computational model of ArchOS and defined a client interface. The ArchOS computational model was designed to provide a high level of modularity and maintainability for both the designer of real-time applications using ArchOS, and for the ArchOS implementors themselves. The ArchOS client interface defines the client's view of the total functionality of ArchOS. All of the ArchOS primitives are also described at this level and internal system mechanisms are also being developed. Third, along with the development of the client interface, we also investigated a specification technique for ArchOS. In this work, such questions as how to convert the precise needs of the client into the system resources which will satisfy these needs, and how to describe a system which can provide these resources were studied. Fourth, we created the System Functionality Specification for ArchOS (CDRL Item No. A002). This specification describes Archons' guiding concepts and provides an overview of ArchOS functionalities. Finally, the basic Archons interim testbed facility is now stable, and we are ready to build a basic kernel to support further experiments.

During 1985, we have started to design the ArchOS system architecture based on the previous specifications. In particular, we substantially used our arobject model to build ArchOS facilities.

---

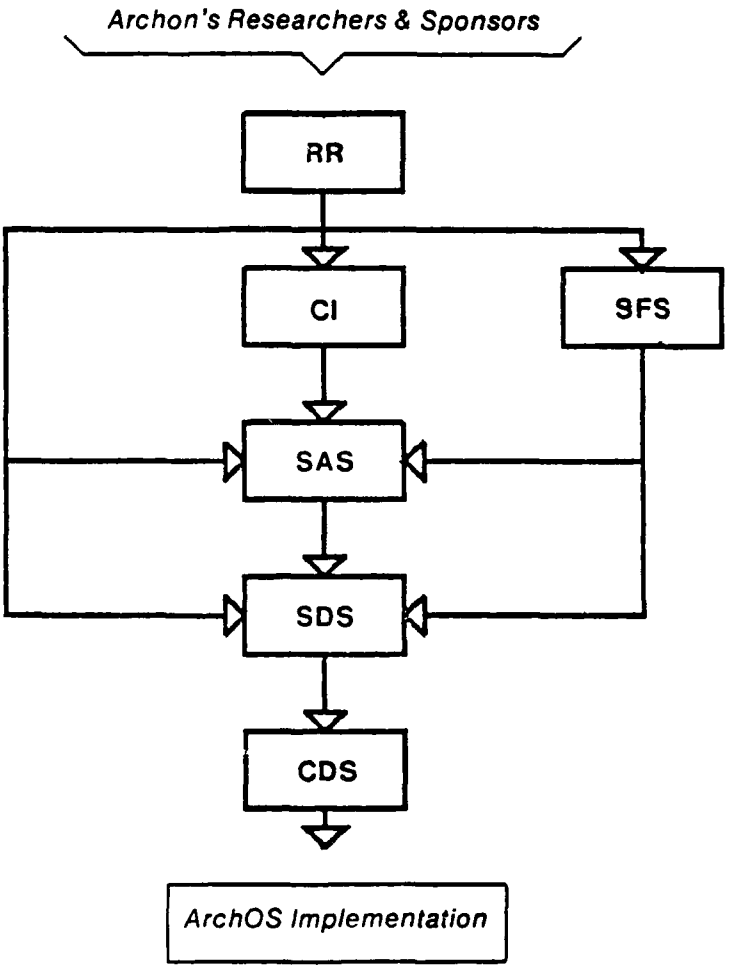
[2] UNIX is a trademark of AT&T Bell Laboratories

1.2 Relationship to Previous Specification Documents

This document contains two reports which describe the system architecture specification (SAS) of ArchOS and the system design specification (SDS) of ArchOS.

As we proposed in [Jensen 83b], we have decided to adopt a top-down approach to the design and to produce a series of specification documents as we refine our design of the ArchOS operating system. Figure 1-1 shows the planned series of documents and their relationships to one another.

The first three documents, namely research requirements (RR), system functionality specification (SFS) and client interface specification (CI), are complete and are included in the "Functional Description of ArchOS", [Jensen 85] (CDRL Item No. A002). This document contains the next two reports in the design sequence, namely the system architecture specification (SAS) and the system design specification (SDS), summarizing the internal system architecture of the ArchOS operating system.



RR: Research Requirements
CI: Client's Interface Specification
SFS: System Functionality Specification
SAS: System Architectural Specification
SDS: System Design Specification
CDS: Component Design Specification

Figure 1-1: ArchOS Development Steps

2. ArchOS System Architecture Description

This chapter describes the system architecture of ArchOS which supports all of the facilities and primitives described in the client interface document [Jensen 85]. Since we view "ArchOS system architecture" as a high-level view of the internal system design, this chapter focuses on the description of the major components of ArchOS operating system and leaves the detailed design to the next chapter.

From a conceptual point of view, ArchOS consists of the ArchOS kernel and system arobjects which are grouped together to form *subsystems*. The ArchOS kernel provides a set of basic mechanisms which can support arobjects in both kernel and client address spaces. An ArchOS subsystem consists of one or more system arobjects which provides an ArchOS facility. In other words, the ArchOS primitives described in the client interface document are defined as a set of operations of these system arobjects.

2.1 Overview

ArchOS is not a time-sharing operating system nor a network operating system for a set of workstations. ArchOS is a physically dispersed operating system which performs decentralized system-wide resource management for a *decentralized computer* which can be physically dispersed across 10^{-3} to 10^{3} meters, interconnected without the use of shared primary memory.

We view ArchOS as a reserach vehicle to perform research on the issues of decentralization in real-time distributed systems at the OS level and below. Thus, the ArchOS facilities were designed to allow test applications to be constructed with which to study ArchOS characteristics, but they need not provide a particularly complete set of facilities in an application production environment. As we specified in the client interface document, some system facilities are not fleshed out, but each facility is functionally closed so that each function can be evaluated with respect to our research issues.

The basic model for system-wide resource management in ArchOS is based on our previous conceptual/theoretical study and experimental study. In particular, we are interested in decentralized resource management schemes where each global decision is made *multilaterally* by a group of peers through negotiation, compromise, and consensus. Thus, ArchOS facilities are provided by a group of cooperating servers of which each service entity is built by an arobject on each node. Since each arobject can have its own state and can activate computation independently of other arobjects, cooperation among servers can be easily represented by a set of arobjects.

Furthermore, an arobject cannot fetch or modify the status of any other arobject without invoking its corresponding operation at the target arobject, thus the modularity of arobjects can also be achieved as well as potential parallelism between the caller and called arobjects.

From the higher-level point of view, the overall structure of ArchOS is divided into two system components. One component is an ArchOS kernel which provides a set of basic mechanisms and links the higher-level policy with the mechanisms and creates an arobject environment. The other is a set of system arobjects which implements ArchOS system facilities. A system arobject can be further distinguished by where it resides, namely a kernel arobject which exists in the kernel and non-kernel arobjects which exist outside the kernel.

Since an ArchOS facility is built based on cooperating arobjects, it is easy to change the internal implementation without impacting client programs. Furthermore, certain facilities are also built based on the notion of policy/mechanism separation [Wulf 81], so that an existing policy can be changed or a new type of policy can be created without changing mechanisms. Thus, ArchOS facilities can be easily tuned towards a particular type of application environment without changing the mechanisms in the system.

2.2 ArchOS System Architecture

The system architecture of ArchOS should be flexible enough so that various degrees of decentralized resource management schemes can be applied in ArchOS without incurring major modification cost. Since the Archons project is not attempting to develop a computing facility, we must consider "openess" of the system architecture.

The system structure of ArchOS was designed to improve the robustness, extensibility, and modularity of OS functions. There is no centralized decision maker in the system for any type of system-wide resource management. That is, any system-wide resource management is performed by the cooperation among the essentially identical peer server modules at each node. These modules allow the use of the "most decentralized" algorithms (that are practical) for managing system resources.

The ArchOS system architecture was designed without assuming any specific hardware-level structures such as node architecture and communication architecture. However, to maximize performance, it is preferable to have the following architectural support: 1) Each node should have two or more processors with a reasonable amount of shared memory; 2) Each node can communicate

with another node, a set of nodes (i.e., multicasting), or all of the nodes within a reasonable amount of time; 3) Each node should have a logically homogeneous environment. That is, even if the processor architectures are different, all of the nodes can emulate the ArchOS functionality.

2.2.1 Objectives

The major objectives are derived from the research requirements and the functionality of ArchOS. In particular, ArchOS is designed to be a research vehicle for investigating various decentralized resource management techniques, so that various types of OS facilities can be added, changed or deleted.

The design objectives of ArchOS system architecture are summarized as follows:

- **Open system architecture:**
  Since ArchOS is designed as a research vehicle to investigate various decentralization issues at operating system level and below, a new system component can be easily added as well as deleting the existing facility.

- **Highly decentralized resource management:**
  ArchOS should not have any centralized decision maker in the system and should be structured as essentially identical peer modules replicated at each system node.

- **High system availability:**
  To maintain high system availability in the face of node and communications faults, provide for no lasting degradation of system function or response beyond the actual resource loss encountered.

- **High system modularity:**
  Previous distributed systems have been constructed around the physical communications limitations of the system, as opposed to being based on such modern software engineering principles as abstract data types and information hiding for defining modularity.

- **Highly extensible facilities:**
  To construct highly extensible facilities which will limit the cost of redesign for implementing widely divergent operating system facilities for experimentation with the fundamental concepts of interest to us in this research.

- **Reasonable system performance:**
  To provide reasonable system performance to support applications with real-time constraints. ArchOS must consider time-critical functions for which time constraints define some system failure modes (i.e., issues of timeliness will not be ignored as they are in some systems, nor will they be dealt with by simply providing a large ratio of available to currently used resources, which is how virtually all other real-time systems strive to meet response time requirements).

2.2.2 Basic Approach

A basic approach to meet the previous objectives is that we provide a uniform view of the system components, namely a kernel and a set of arobjects. An ArchOS kernel provides basic mechanisms for ArchOS facilities and the cooperating arobjects support the policies to provide a particular set of services. While we avoid a large monolithic kernel, we try to improve robustness, modularity, and extensibility of each ArchOS facility as well as increasing potential parallelism by using the cooperating arobjects.

To meet the previous objectives, we will use the following approaches:

- **Open system architecture:**
  We will build ArchOS based on an object-oriented architecture view, so that the ArchOS kernel is not a monolithic kernel and consists of a set of basic mechanisms and kernel arobjects. On top of the kernel, ArchOS facilities are provided by cooperating arobjects. Since an arobject encapsulates its associated information (information hiding), key properties of each facility can be easily modified without affecting the other facilities. Furthermore, additional system flexibility will be provided by identifying and separating mechanism and policy in ArchOS facilities.

- **Highly decentralized resource management:**
  Since we try to avoid having any type of centralized decision maker for system-wide resources and each arobject can have its own computational state and knowledge, it is easy to form cooperation among peer modules in the system. Unlike traditional procedure invocation in abstract data types, a caller cannot invoke an operation on an arobject in a *master-slave* manner, but must use a form of rendezvous. Based on its own state and decision, the receiver arobject has the right to accept, reject, or delay the invocation of the requested operation.

- **High system availability:**
  The use of atomic transactions to maintain consistency will result in continuance of useful computation in the presence of lost, delayed, inaccurate, or incomplete information. Also, a service handled by a set of replicated arobjects can be easily supported by using the one-to-many communication capability together with transactions.

- **High system modularity:**
  Since an arobject can encapsulate its state and implementation details from the other arobjects, it is easy to use it as a system building block. System modularity is also achieved not only by using arobjects, but also through policy/mechanism separation in ArchOS facilities.

- **Highly extensible facilities:**
  ArchOS facilities are implemented as a set of system arobjects, thus it is possible to create a new type of service by creating a new set of arobjects and registering them.

- **Reasonable system performance:**
  To provide reasonable system performance, ArchOS subsystems are built based on a set

of arobjects which can be executed on a multiprocessor based node without major redesign.

A conceptual view of the ArchOS system architecture is depicted in Figure 2-1. On each node built from one or more processors with shared memory, there are ArchOS kernel and system arobjects.

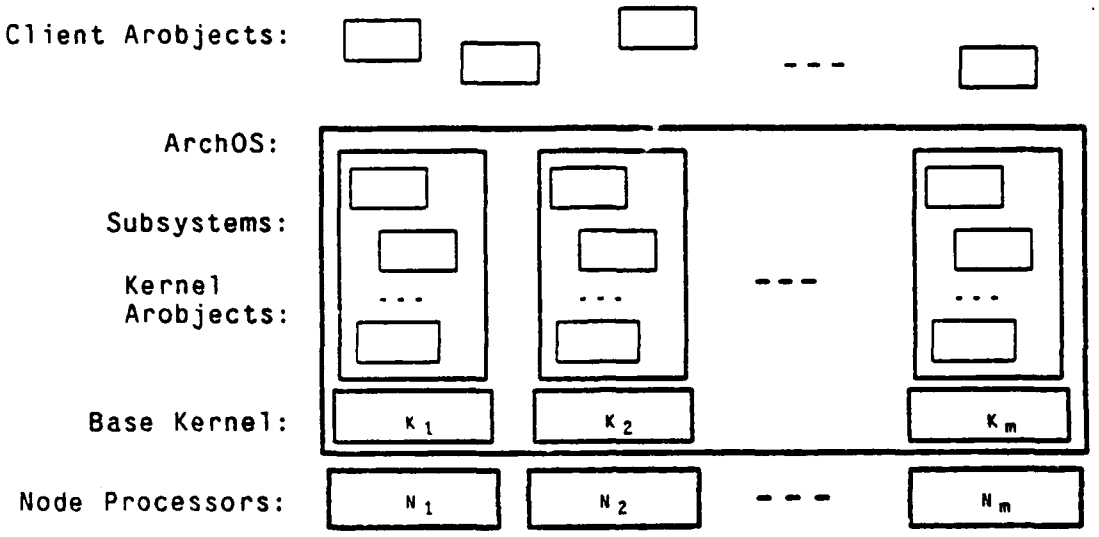


Figure 2-1: Overview of ArchOS System Architecture

The *ArchOS kernel* consists of a set of basic mechanisms and a set of system arobjects which reside within a kernel domain. No system arobjects can be accessed without invoking the ArchOS primitives. A client process must use a communication primitive, *request*, to invoke any system primitives. ArchOS primitives defined in the client interface are provided either as *kernel* primitives or *system* primitives. Kernel primitives are defined as an operation on kernel arobjects, while system primitives are provided by an operation on the system arobject which exists above the kernel.

ArchOS Subsystems are one or more arobjects grouped together by sharing the same responsibility for providing a particular *service*, namely a set of functions in an ArchOS facility.

A *system arobject* can be a kernel arobject or an ordinary arobject which resides in the user domain. A *client arobject* can be created on top of the kernel and resides in the user domain.

2.2.3 Functional Dependency among ArchOS Subsystems

The relationship among ArchOS subsystems can be summarized by abstracting out the detailed interaction among arobjects across the various subsystems. Since each subsystem consists of a set of cooperating arobjects, many activities are initiated by invoking a particular operation in the arobject. This section focuses on the functional dependency among ArchOS subsystems, rather than on the detailed interaction among arobjects. The description of the internal structure of each subsystem will be given in the next chapter.

From a functional point of view, ArchOS subsystems can be depicted as in Figure 2-2.

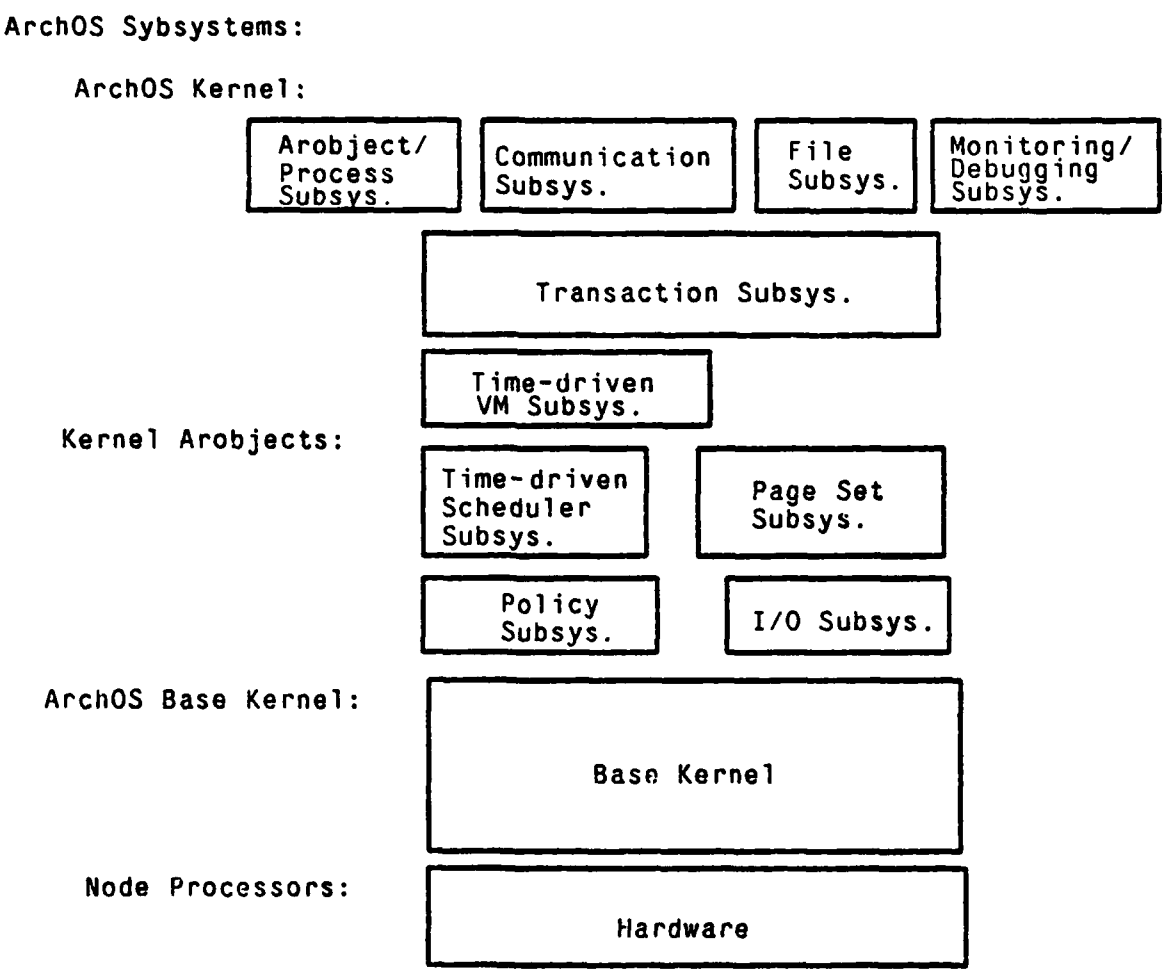


Figure 2-2: Functional Dependency among ArchOS Subsystems

In Figure 2-2, each box represents a subsystem in ArchOS. However, unlike a traditional hierarchical layered structured system, an operation at a lower-level subsystem can be directly invoked from a client arobject.

The *ArchOS Kernel* provides a set of basic mechanisms and consists of a set of system arobjects which reside within a kernel domain. The portion of the basic mechanisms is called the *Base* kernel. Unlike a traditional monolithic kernel, the ArchOS kernel contains a set of of system arobjects, which we refer to as *kernel arobjects*, and it can initiate independent computation from client arobjects. A certain set of basic mechanisms are realized to coordinate with the policy definition module, so that facilities may provide a different type of functionality to clients.

The *Policy management subsystem* maintains a user-definable system module, called the *policy definition module* which consists of a *policy body* and a set of *policy attributes*. Since a policy definition module can be placed in the kernel, a system arobject, or a client arobject, the policy management subsystem creates and destroys a *policy definition descriptor* which indicates the location of the policy definition and the information related to the policy body and attributed set.

The *Page set subsystem* provides a uniform view of secondary storage, namely a *page set* in ArchOS. An arobject can access a file through the page set and logical disk subsystems. On each node, there is a page set manager which can allocate or deallocate a permanent type or atomic type of page set on a specified logical disk.

The *I/O device subsystem* manages interaction between I/O devices and arobjects. Any signal from I/O devices is handled at this subsystem and is translated into an invocation request for a system arobject.

The *System monitoring and debugging subsystem* provides various abilities to monitor and control the behavior of cooperating arobjects and their processes during execution.

The *Time-driven scheduler subsystem* provides a time-driven scheduling facility which may be altered by adding, changing or selecting a different type of scheduling policy. The basic mechanism is based on ArchOS' "best effort" scheduling model [Locke 85]. This subsystem also provides functions such as obtaining time information and setting/resetting scheduling parameters for clients.

The *Time-driven virtual memory subsystem* provides a virtual memory management facility in a time-driven fashion. The time-driven scheduler may be called from this subsystem to initiate pre-paging in/out activities in such a way that the system can execute time-critical tasks in a timely manner.

The *Transaction subsystem* provides transaction mechanisms for operations on arbitrary types of arobjects. This subsystem supports two types of transactions, namely *elementary* and *compound*

transactions which can be nested in arbitrary combinations. To coordinate an atomic update for a transaction, the transaction subsystem must cooperate with one or more page set managers which the transaction has visited.

The *Arobject/process management subsystem* provides the basic functions to create and destroy arobjects and processes. This subsystem also manages binding and unbinding of arobjects/processes *reference* names and supports *binding protocols* to match a requestor with one or more suitable server arobjects. The requestor arobject invokes an operation by using the *invocation protocol* through the communication subsystem.

The *Communication subsystem* provides the *invocation protocol* and manages arobject invocation among arobjects. This subsystem provides not only *one-to-one* communication for a conventional client-(single) server model, but also *one-to-many* communication for the client-cooperating multiple server model.

The *File subsystem* provides system-wide location-independent file access. Although the file subsystem does not support a hierarchical file name space, a system-wide flat name space is maintained at each node. The file subsystem also provides three kinds of file properties. A *normal* file has the data portion of the file arobject in volatile storage. A *permanent* file's data portion is allocated in permanent (non-volatile) storage and an *atomic* file has its data portion declared as an atomic data object. For atomic files, the transaction facility is also supported through the transaction subsystem.

2.3 Structure of ArchOS Subsystems

An ArchOS subsystem consists of a set of system arobjects, called *components*, resident on each node and provides a system-wide or local service within a node. Each subsystem was designed based on a generic server structure for ArchOS in order to provide reliable and fast service of the requested system functions.

2.3.1 Objectives

The internal structure of a subsystem should meet the following objectives:

- Various types of decentralized resource management schemes should be easily implemented in a subsystem.
- A subsystem should be able to serve concurrent requests efficiently, fairly and without deadlocks.
- A subsystem should be able to support replicated arobjects to increase availability of its service.

- Each component should be able to stop independently and resume its activity without major disturbance.
- Within a component, concurrent operation invocation should be easily provided if the service requests can be handled simultaneously.

2.3.2 Internal Structure of Subsystems

The internal structure of subsystems is based on a generic server model which can provide system-wide service in a reliable manner. The server model was used to provide the template for different types of servers and to reduce the design complexity of subsystems.

The server model consists of *service protocols*, which define a complete interaction between a client and all servers providing the same service, and a *server structure*.

Our view of system service at the subsystem level is depicted in Figure 2-3.

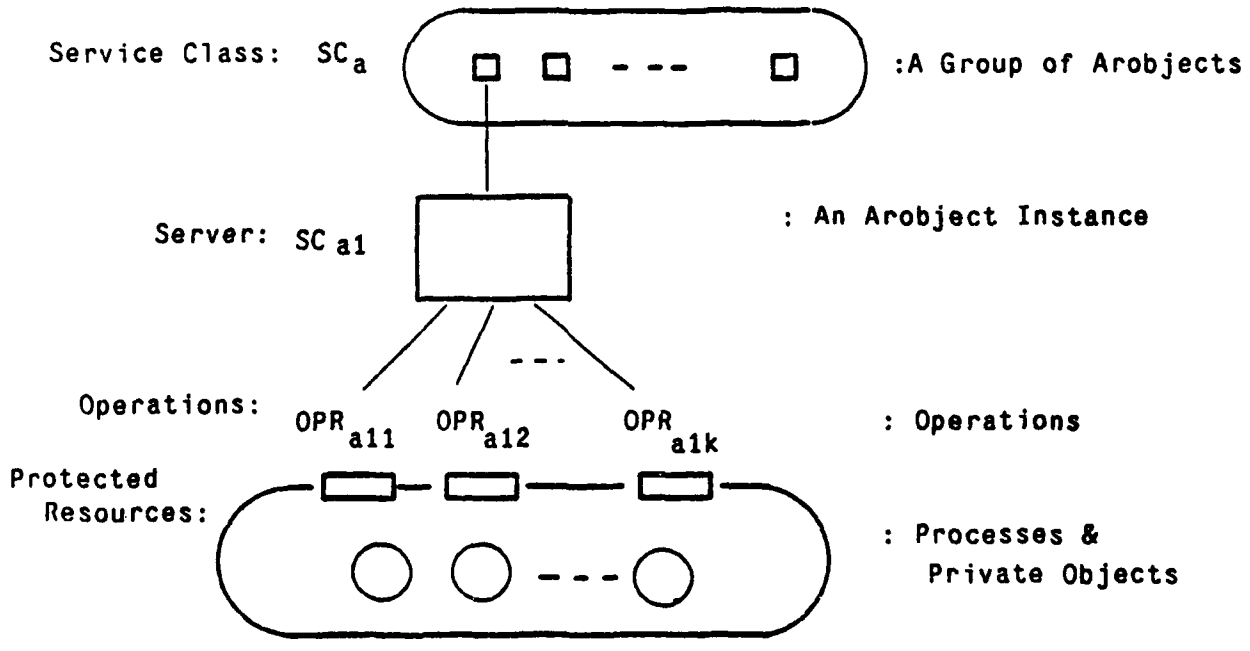


Figure 2-3: Relationship among Service, Server, and Arobject

At the subsystem level, a *reference name* is given to each *service class* which provides a particular service, namely a set of functions for an ArchOS facility. A service class is implemented by a set of

service instances, namely *servers*. A server is formed by an instance of an arobject and is normally replicated at each node. However, a different set of servers can also form a service class, since a reference name can be shared by these servers.

In general, services are replicated on each node, and system-wide resource management is performed by using the service protocol to define interaction between client and server as well as inter-server communication.

2.3.2.1 Service Protocol

A subsystem provides a service or a set of services for a client. A client can locate a suitable server for a requested service and get the result back by initiating the service protocol. The service protocol consists of three protocols: *binding*, *invoking*, and *inter-server* protocols.

The *binding* protocol defines how a client determines a suitable server for a requested service from a set of servers. For instance, we can apply the following binding policies for certain types of resource management such as creating a new instance of an arobject or a process in the system:

- **First Fit (FF):**
  The first server from the matched servers will be selected to submit a service request.

- **Random Fit (RF):**
  A server from the random selection will be used to submit a service request.

- **Best Fit (BF):**
  One of the best matched servers will be used to submit a service request.

- **Best Effort Fit (BEF):**
  The best matched server according to the best-effort (decision making) algorithm will be used to submit a service request.

It should be noted that RF protocol can be used without any interaction with potential servers, while FF protocol may need at least one reply from the servers, BF needs all replies, and BEF needs all or some replies.

The *invoking protocol* defines how a client can invoke an operation at a specific or non-specific server(s) and can get the service result. In ArchOS, a client arobject can invoke an operation synchronously as well as asynchronously. Thus, it is possible for a client to invoke an operation at multiple servers simultaneously.

- **synchronous/specific server request:**
  A client can invoke an operation on a specific arobject by using a *Request* primitive.

- **synchronous/non-specific servers request:**
  A client can invoke an operation on a specific service class by using a *RequestAll* primitive followed by the necessary *GetReply* primitives.

- **asynchronous/specific server request:**
  A client can invoke an operation on a specific arobject with blocking itself by using a *RequestSingle* primitive.

- **asynchronous/non-specific servers request:**
  A client can invoke an operation on a specific service class by using a *RequestAll* primitive.

The *Inter-server protocol* defines how an individual server can exchange control or status information to perform a service. Thus, the inter-server protocol is dependent upon the nature of the service. For instance, local resource information of a server can be distributed to all other servers periodically by invoking a suitable operation for a service class. This type of multicasting can also be performed by using a *requestall* primitive.

2.3.2.2 Generic Structure for a Server

The generic structure for a server is based on the nature of service the server provides and the characteristics of an arobject. Since any system service should be highly available, a server itself must increase its availability by avoiding necessary blocking periods and deadlocks.

Each arobject has at least one process which has responsibility to create the necessary task force to provide a service. The number of processes may be created dynamically depending on the request load. On the other hand, a certain server might have a fixed number of processes to handle a fixed number of tasks. Thus, the server structure is related to the nature of the service which is provided by the cooperating arobjects.

We can summarize the generic structure for a server in Figure 2-4.

- Type I: *a single process*
  The initial process accepts, requests, and processes them one at a time. That is, the execution order of the operations will be totally serialized. Each operation invocation will be handled by a corresponding procedure or function within the initial process.

- Type II: *functionally partitioned workers*
  The initial process creates a number of workers according to the number of operations. Coordination between workers is handled by the workers. Each operation invocation is accepted by a specific worker process assigned to do the task. Thus, a worker issues the *Accept* primitive to wait for a specific type of requested operation.

- Type III: *replicated workers*

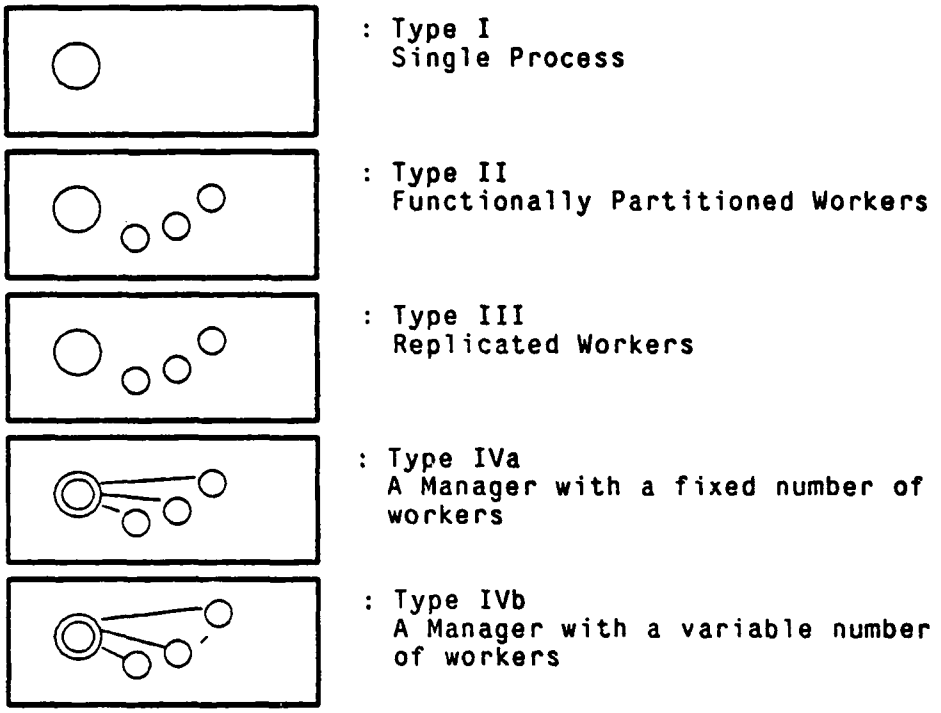


Figure 2-4: Generic Server Structures for a Server

The initial process creates a number of replicated workers to accept "any" operation. Coordination between workers is handled by the workers. Workers can be placed at different nodes, so parallel execution of requested operations might be possible. Each worker issues an *AcceptAny* primitive to wait for an invocation request.

- Type IVa: *a manager with a fixed number of workers*
  The initial process creates a fixed number of workers as a service task force and becomes a manager of them. Coordination between workers is handled by the initial process and each worker may or may not be assigned the same task.

- Type IVb: *a manager with variable number of workers*
  The initial process acts as a manager of workers and accepts all incoming invocation requests and creates necessary workers on demand. Coordination between workers is handled by the initial process and each worker may or may not be given the same functionality.

3. ArchOS System Design Description

This chapter contains the ArchOS system design description and describes the internal architecture of each ArchOS subsystem. From a conceptual point of view, the ArchOS operating system consists of the ArchOS kernel and a set of subsystems composed of a set of cooperating system arobjects. While the ArchOS system architecture description in the previous chapter discussed higher-level views of the system structure, this chapter focuses on how each subsystem is built based on cooperating arobjects.

3.1 Overview

The ArchOS system design description discusses the internal architecture of ArchOS focusing on the ArchOS kernel and subsystems. The ArchOS kernel provides basic mechanisms for ArchOS facilities and each subsystem implements each service facility of ArchOS.

As discussed in the previous chapter, the ArchOS kernel consists of two layers: the lower layer is the *Base* and the higher layer is a setof *kernel arobjects*. On top of the base kernel, a kernel arobject can be created to increase the concurrency within the base kernel. ArchOS provides a system-wide arobject environment on top of the ArchOS kernel. Since ArchOS facilities are implemented by cooperating system arobjects, they can be modified, added, or deleted without having a major cost.

A subsystem consists of a set of cooperating arobjects which can provide the actual services to clients. In general, each *component* of the subsystem is replicated on each node, so a system-wide service is provided as the result of interaction among components. Each component also has the responsibility to recover from so-called "clean and soft failure" [Bernstein 83] at a node.[3]The chosen recovery sequence maintains the consistent state of the server.

In the following sections, we describe the ArchOS kernel and each ArchOS subsystem focusing on its internal structure, key algorithms, and inter-server protocols.

---

[3]We limit our discussion of resiliency against "clean and soft" failure in which some of nodes of the system simply stop running and loose the contents of main memory, but the contents of stable storage used by the failed nodes (computers) remain intact.

3.2 ArchOS Kernel

3.2.1 Overview of ArchOS Kernel

The ArchOS kernel is a collection of low-level mechanisms and kernel arobjects which provides basic service facilities to support the ArchOS operating system. In order to increase its flexibility, the ArchOS base kernel only provides time critical system functions such as time-driven scheduler, kernel memory manager and kernel arobject/process manager. The other low-level OS functions such as transaction subsystem, arobject/process subsystem, communication subsystems can be created on top of the base kernel. Further more, a certain type of system function such monitoring/debugging subsystems are implemented by system arobjects which can be created on top of the ArchOS kernel. As a result, ArchOS primitives defined in the client interface are implemented either as a *kernel* primitive, an operation on kernel arobjects, or an operation on system arobjects.

Internally, the ArchOS kernel consists of a *base* kernel and a set of kernel arobjects which can create a kernel arobject environment. Even though the base kernel will be built as a collection of independent low-level modules, an object-oriented view must be maintained. It is a part of our study to explore the minimum structure of the base kernel as well as the efficient structure the base kernel.

Figure 3-1 shows logical structure of ArchOS Kernel.

3.2.2 ArchOS Base Kernel

The ArchOS base kernel provides basic mechanisms for ArchOS facilities, thus creating a uniform arobject environment for system arobjects. Since the ArchOS kernel is not a traditional monolithic kernel, a set of kernel arbojects are running on the top of the base kernel and supporting necessary low-level functions for ArchOS facilities.

The major functionality of the base kernel can be summarized as follows:

- Creation and destruction of kernel arobjects and processes
- Communication between kernel arobjects
- Dispatching mechanism for the time-driven scheduler
- Address space management for time-driven virtual memory manager
- Low-level synchronization between processes and interrupt/trap

In other workds, the basic kernel is responsible to perform local resource management for creating

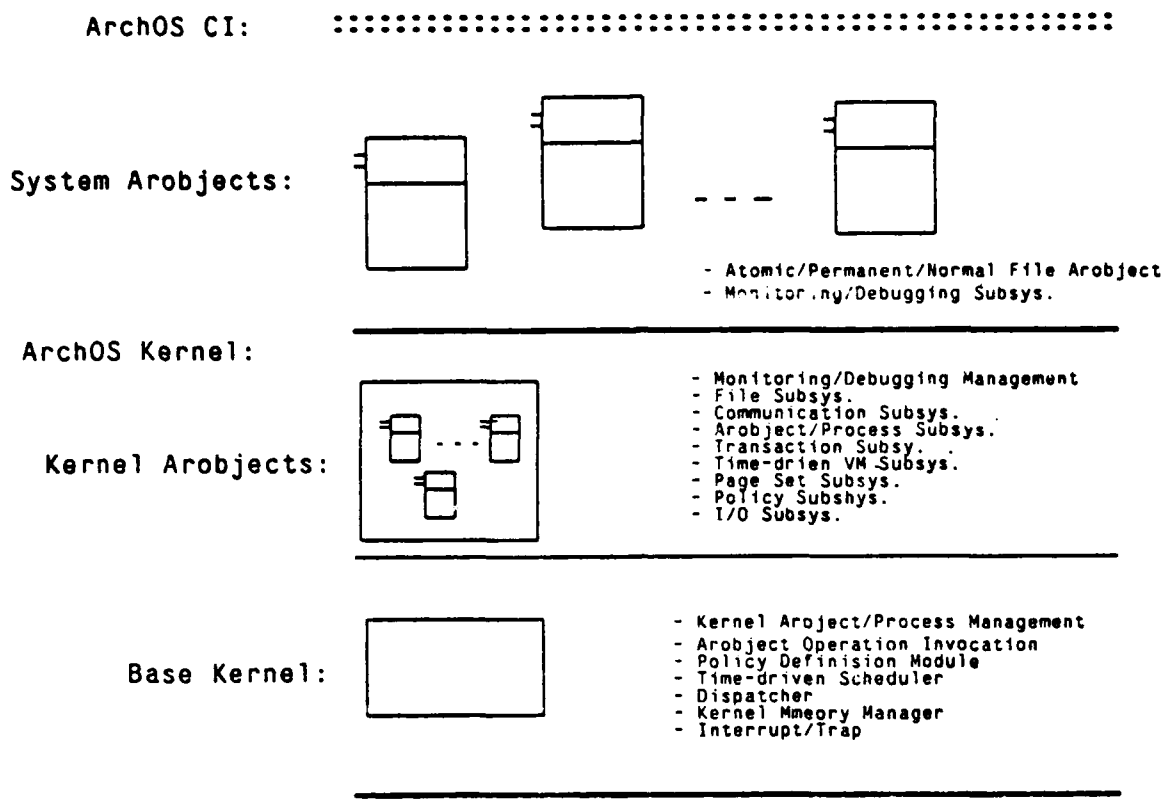


Figure 3-1: Logical Structure of ArchOS Kernel

and destructing kernel objects and for invoking its operations. Any remote or system-wide resource management decisions are decided based on coordination among kernel arobjects.

3.2.2.1 Kernel Arobjects and Processes

The ArchOS base kernel provides creation and destruction of a kernel arobject in its node. A kernel arobject is similar to a normal arobject except that it resides in a kernel address space and shares the address space among the other kernel arobjects. Similarly, all of the kernel processes share their address space even though each process logically belongs to a specific kernel arobject.

Binding of a kernel arboject/process and a reference name is also managed in the same way. Unlike a client's reference name, a kernel arobject's reference name is treated as a "well-known" name representing a service class.

The following normal ArchOS primitives are used to manage kernel arobjects and processes.

```
arobject-id = CreateArojbect(arobj-name [, init-msg])
process-id = CreateProcess(process-name [, init-msg])
val = KillArobject(aid)
val = KillProcess(pid)

aid = SelfAid()
paid = ParentAid(aid)
pid = SelfPid()
ppid = ParentPid(pid-x)

val = BindArobjectName(aid, arobj-refname)
val = BindProcessName(pid, process-refname)
val = UnbindArobjectName(aid, arobj-refname)
val = UnbindProcessName(pid, process-refname)

aid = FindAid(arobj-refname)
pid = FindPid(process-refname)
aid-list = FindAllAid(arobj-refname)
pid-list = FindAllPid(process-refname)
```

AID arobject-id The unique identification of the instantiated arobject.

PID process-id The unique identification of the instantiated process.

AROBJ-NAME arobj-name
The name of arobject to be instantiated.

PROCESS-NAME process-name
The name of process to be instantiated.

MESSAGE *init-msg
A pointer to the initial message which contains initial parameters for the INITIAL process.

AROBJ-REFNAME arobj-refname
The requested reference name for an arobject given by aid.

PROCESS-REFNAME process-refname
The requested reference name for a process given by pid.

AID-LIST aid-list The list of corresponding aid's.

PID-LIST pid-list The list of corresponding pid's.

AROBJ-REFNAME arobj-refname
The reference name of related arobject(s).

PROCESS-REFNAME process-refname
The reference name of related process(es).

It should be noted that the above primitives are identical to the ordinary ArchOS primitives clients can use, but only local creation and destruction of kernel arobjects and processes are supported in the ArchOS kernel. When an instance of a kernel arobject or process is created, an ordinary arobject or process descriptor will be created and managed uniformly at the kernel. (The description of the arobject/process descriptor is given in Section AROBJECT).

3.2.2.2 Kernel Communication Management

The ArchOS base kernel provides a local communication mechanism among kernel arobjects. A kernel arobject can invoke an operation at a single destination arobject or at multiple arobjects providing the same service name (i.e., sharing the same reference name).

---

trans-id = **Request**(arobj-id, opr, msg, reply-msg)
trans-id = **RequestSingle**(arobj-id, opr, msg)
trans-id = **RequestAll**(arobject-name, opr, msg)

pid = **GetReply**(trans-id, reply-msg)

(trans-id, requestor, opr) = **AcceptAny**(opr, msg)
(trans-id, opr) = **Accept**(requestor, opr, msg)

ptr-mds = **CheckMessageQ**(qtype, requestor, opr, req-trans-id)

trans-id = **Reply**(pid, req-trans-id, reply-msg)

TRANSACTION-ID trans-id
: The transaction id of the transaction on whose behalf the request is being made.

AID arobj-id
: The unique id of the receiving arobject.

OPE-SELECTOR opr
: The name of the operation to be performed.

MESSAGE *msg
: A pointer to the message which contains the parameters of the operation to be performed. The message to the destination arobject must not contain any pointers (i.e., call-by-value semantics must be used).

REPLY-MSG *reply-msg
: A pointer to the reply message.

MSG-DESCRIPTORS *prt-mds
: Pointer to a list of the message descriptors selected by the specified selection criteria.

MSG-Q qtype This indicates either "request-" or "reply-" message queue.

AID requestor The aid of the requesting arobject.

OPE-SELECTOR opr
The operation to be performed. The "opr" parameter can be a specific operation name or "ANYOPR".

TRANSACTION-ID req-trans-id
The transaction id of the corresponding *RequestSingle* or *RequestAll* primitive.

3.2.2.3 Policy Management

The policy management provides system functions to add, delete, and modify the policy definition module in ArchOS. Since the placement of the *policy definition module* is a major issue in terms of the system performance, ArchOS allows a client to specify the location by using a *policy definition descriptor*. The policy definition module consists of a *policy body* and a set of *policy attributes*. Both the policy body and attributes can be modified at runtime.

---

val = **SetPolicy**(policy-name, policy-def-desc)
val = **SetAttribute**(policy-name, attribute-name, attribute-value)

pdd = **AllocatePDD**()
val = **FreePDD**(pdd)

BOOLEAN val TRUE if the specified policy was set properly; otherwise FALSE.

PDD policy-def-desc
A pointer to the policy definition descriptior.

ATTRIBUTE-NAME attribute-name
The name of attribute to be set.

ATTRIBUTE-VALUE attribute-value
The actual value for the attribute.

The *SetPolicy* primitive links a user-defined policy definition module to the ArchOS base kernel. The *SetAttribute* primitive set a specific value(s) for its one of attribute. Since a policy definition descriptor is maintained in the base kernel, any access to the actual policy body or a value of its attribute can be easily made.

The *AllocatePDD* primitive allocates a policy definition descriptor in the base kernel and *FreePDD* releases the allocated descriptor.

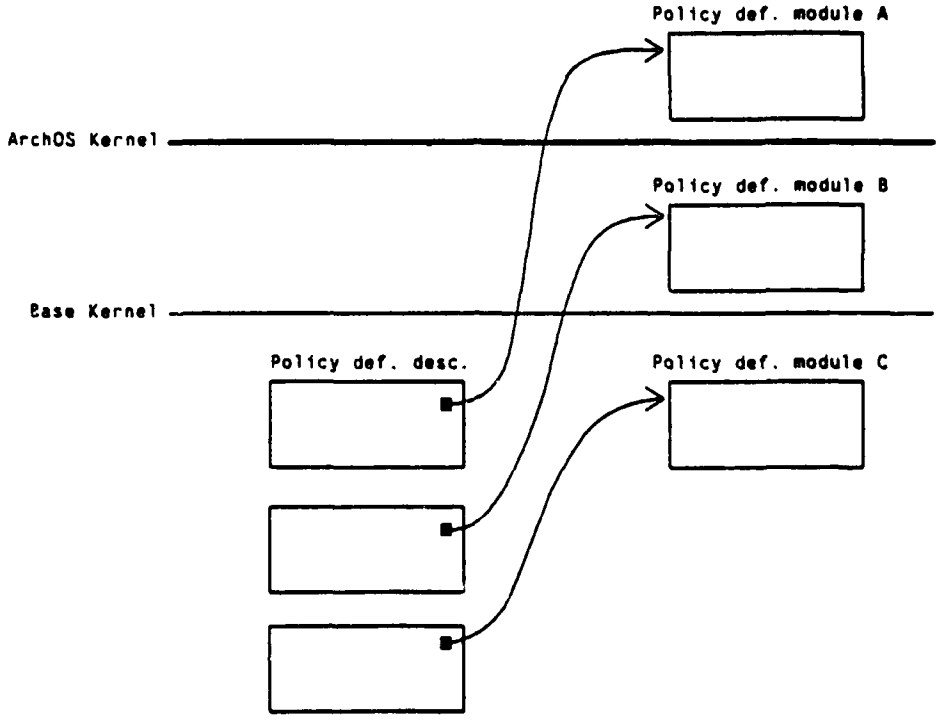


Figure 3-2: Policy Definition Module and PDD

3.2.2.4 Time-driven Scheduling Management

The interface between the scheduler and the remainder of the ArchOS kernel is a simple one in which the scheduler acts as a simple object providing operations and maintaining its own scheduling data base, including its scheduling queue. These operations, of course, will not be available to the ArchOS client, but will be used internally by ArchOS to generate scheduling requests whenever needed. The operations defined by the scheduler are as follows.

pid-list = **Schedule()**

pid-list = **RequestSwapList()**

SetScheduleInfo(pid)

Deschedule(pid)

PID-LIST pid-list The list of pids.

PID process-id The unique identification of the instantiated process.

The *Schedule()* primitive returns a list of the process ids (pids). The first pid indicates a process to be executed at this time and the following pids are candidates for the following scheduling point. No parameters are passed, and the scheduler makes its decision directly from the information within its data base.

The *RequestSwapList()* primitive returns a list of the pids which indicates candidates for the following swapping decisions at the virtual memory manager level.

The *SetScheduleInfo* primitive enters the necessary scheduling information for the specified process to the scheduler. The scheduler places the process pid and all its scheduling parameters into its database and prepares for making the scheduling decision required when the next **Schedule** operation is performed. This decision-making process operates continuously, concurrently with the application processing, preparing its next scheduling decision.

The *Deschedule* primitive removes a process pid from its queue, updating its current scheduling database to prepare for the next *Schedule* operation.

The Scheduler will use the interrupt mechanism in the processor running the ArchOS kernel to invoke other kernel mechanisms when it makes relocation decisions or must make decisions regarding process scheduling with respect to other nodes. In addition, the interrupt mechanism will be used to interrupt the kernel and application processing when sufficient time has elapsed that the current scheduling decision must be reconsidered.

3.2.2.5 Address Space Management

The base kernel provides a number of primitive, though powerful mechanisms for constructing, manipulating, and accessing the definitions of the virtual address spaces of processes. A process' virtual address space (or simply *address space*) is illustrated in Figure 3-3. An address space consists of four non-overlapping *regions*:

1. **Kernel Region:**
   The Kernel Region is identical for all processes, and is only accessible when executing in kernel mode. It contains all of the kernel code and data structures. The kernel is shared by all processes and (at least in the initial version of ArchOS) will be non-pageable.

2. **Private Region:**
   The Private Region is unique for each process, and it consists of five *segments*: Kernel

Stack Segment, User Stack Segment, User Heap Segment, User Data Segment, and User Text Segment. Each of these segments will be discussed below.

3. **Shared Region:**
   The Shared Region is unique for each arobject, but shared by all processes within an arobject. This region contains all of the arobject's private abstract data type instances, along with the code for accessing and manipulating them. It consists of a variable number of segments, one (or more) for each instance of a shared abstract data type, plus a Shared Text Segment and Shared Headers Segment. Each of these different types of segments will be discussed below.

4. **Kernel Interface Region:**
   The Kernel Interface Region is quite small, and primarily contains the code and data areas needed for switching between user and kernel modes. This region is shared by all processes and is non-pageable.

The Kernel Region and Kernel Interface Region are of fixed size, and the virtual to physical address maps for these regions, as well as their protection attributes, do not vary from address space to address space (process to process), or during the lifetime of an address space. As a result, these two regions are constructed automatically whenever a new address space is created, and never altered thereafter. The remaining portion of an address space is comprised of the Private Region and Shared Region. The relative sizes of these two regions can vary from arobject to arobject. However, within an arobject, all processes share the same Shared Region, and hence all processes within a single arobject will have identically sized Private and Shared Regions.

The Private Region and Shared Region each consist of a number of (non-overlapping) segments, of varying sizes and types. The different types of segments have corresponding protection attributes. Some segments are required for each address space, while others are optional. Some segments can be expanded (assuming they have space to grow), while others have a fixed size once allocated. The relative locations of the various segments are somewhat constrained, and as a result, the order in which the segments can be (dynamically) allocated is similarly constrained. Each allocated segment has an associated page set, which is used as the paging area for that segment (see Section 3.7 for a description of page sets). The type of page set associated with a segment (*temporary*, *permanent*, or *atomic*) depends upon the type of the segment.

Private Region Segments

The Private Region contains five distinct segments, whose relative locations must be as shown in Figure 3-3:

Kernel Stack Segment:

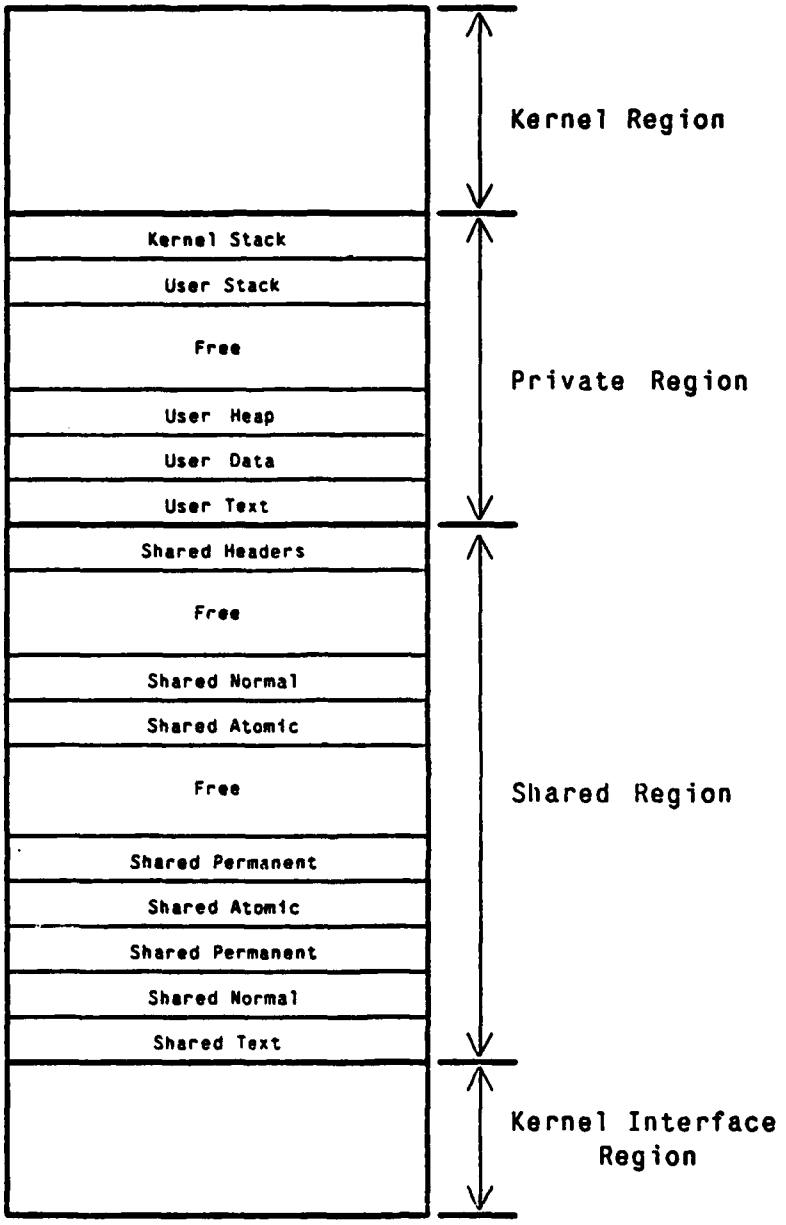


Figure 3-3: Virtual Address Space (One per Process)

The Kernel Stack Segment has a fixed size, which is the same for all address spaces, and is always located at the top of the Private Region. This stack is only accessible while in kernel mode, and it is "substituted" for the User Stack (see below) as part of the operation of switching to the kernel. Since the kernel is not pageable, the Kernel Stack Segment is also not pageable. Like the Kernel Region and the Kernel Interface Region, the Kernel Stack Segment is constructed automatically whenever a new address space is created, and never altered thereafter.

User Stack Segment:

The User Stack Segment is a required segment, and is located immediately below the Kernel Stack Segment. It can vary in size from address space to address space, and during the lifetime of an address space. The User Stack Segment is the only segment in the Private Region which grows downward. All other segments are either fixed size or grow upward to include larger virtual addresses. The User Stack Segment contains the subroutine stack (including parameters and local variables) while a process is executing in user mode. Hence, this segment must be both readable and writable from user mode. The User Stack Segment is pageable and should be associated with a *temporary* page set, which is to be used as the paging area.

User Text Segment:

The User Text Segment is also a required segment, and it is located at the bottom of the Private Region. It contains the code to be executed by the process, and hence its protection is set to allow only execute access while in user mode. The User Text Segment can vary in size from address space to address space, but once allocated it never changes.[4] This segment is pageable and should be associated with the *permanent* page set which contains the code for the process. Note that since the User Text Segment is not writeable, only "page-ins" (and no "page-outs") will ever be required. Furthermore, in order to reduce primary memory requirements, the Time-Driven Virtual Memory Subsystem (see Section 3.10) will arrange for User Text pages to be shared among all processes executing the same code on a single node.

User Data Segment:

The User Data Segment is not strictly required, but it will almost always be present. It is located immediately above the User Text Segment, and should be allocated after the User Text Segment has been specified. The User Data Segment contains the "global", initialized variables that are used by (and private to) the process. It is both readable and writable while in user mode. The User Data Segment can vary in size from address space to address space, but once allocated it never changes.[5] This segment is pageable and should be associated with the *permanent* page set which contains the initialized data for the process. However, only (initial) page-ins will ever be performed using this page set, so as not to destroy the definitions of the *initial* values. All page-outs and subsequent page-ins

---

[4] Due to limitations in the language compilers and linkers, the User Text Segment may contain the code for the entire arobject, rather than just the code required by a single process. In that case the User Text Segments for all processes within a single arobject will have the same size.

[5] As in the case of User Text Segments, limitations in the language compilers and linkers may cause the User Data Segments for all processes within a single arobject to have the same size.

will be to the same *temporary* page set used as the paging area for the User Stack Segment.[6]

User Heap Segment:

The User Heap Segment is not strictly required, but it too will almost always be present. It is located immediately above the User Data Segment, and should be allocated after the User Text Segment and User Data Segment have been specified. The User Heap Segment contains the dynamically allocated variables that are used by (and private to) the process. It is both readable and writable while in user mode. The User Heap Segment can vary in size from address space to address space, and during the lifetime of an address space. Note that since the User Heap Segment grows upward while the User Stack Segment grows downward, expansion of these two segments cause additional space to be allocated from opposite ends of the (single) free area within the Private Region. The User Heap Segment is pageable and should be associated with the same *temporary* page set which is used as the paging area for the User Stack Segment.[7]

Shared Region Segments

The Shared Region contains five distinct types of segments, although there can be multiple segments of a particular type. Also, the placement of those segments within the Shared Region is somewhat more flexible than the placement of the segments within the Private Region. The general structure of the Shared Region is illustrated in Figure 3-3. Note that the entire Shared Region is optional, and will only be present if the arobject definition includes one or more private abstract data type definitions. Also note that the Shared Region is identical for all of the processes (address spaces) which belong to a single arobject, and are resident on a particular node. As a result, the Shared Region should only be modified in a single address space on a given node, in order to have all of the related address spaces on that node updated simultaneously.

The types of segments that can appear in the Shared Region are the following:

Shared Text Segment:

There is a single Shared Text Segment, and it must be present whenever the Shared Region exists. It is located at the bottom of the region, and contains the code which defines the permitted operations

---

[6] Multiple segments can use the same page set for paging purposes, by mapping the virtual page addresses within the various segments directly onto the same page numbers within the page set, i.e. virtual page *k* gets mapped onto page number *k*. Since segments do not overlap, the corresponding paging areas will not overlap either. See Section 3.7 for more details on the use of the page set facilities.

[7] If desired, a separate (temporary) page set could be used, allowing paging to different disks, or even different nodes.

on the arobject's private abstract data types. The Shared Text Segment permits only execute access while in user mode. However, it is assumed that this code will always be entered "indirectly", by first determining which abstract data type instance is to be operated upon, and then directly invoking the requested operation with the specified instance as a parameter. Furthermore, it is assumed that operations will only be directly invoked on instances which are located on the local node.[8] The Shared Text Segment can vary in size from address space to address space, but for a given arobject, once it has been allocated it never changes. This segment is pageable and should be associated with the *permanent* page set which contains the code for the operations defined on the arobject's private abstract data types. Note that since the Shared Text Segment is not writeable, only page-ins (and no page-outs) will ever be required. Furthermore, in order to reduce primary memory requirements, the Time-Driven Virtual Memory Subsystem will arrange for Shared Text pages to be shared among all processes executing the same code on a single node.

Shared Headers Segment:

There is a single Shared Headers Segment, and it too must be present whenever the Shared Region exists. It is located at the top of the region, and contains the "header" information describing all instances of private abstract data types that have been created within the arobject. This header information includes the node on which each instance is located, and the address(es) of the Shared Normal Segment, Shared Permanent Segment, and/or Shared Atomic Segment associated with each instance. The header information is used whenever operations are to be invoked on specified abstract data type instances, and hence the Shared Headers Segment allows read-only access while in user mode. The Shared Headers Segment can vary in size from address space to address space, and during the lifetime of an address space. However, within a single arobject, all address spaces will have identical Shared Headers Segments. The Shared Headers Segment is the only segment in the Shared Region which grows downward. All other segments are either fixed size or grow upward to include larger virtual addresses. The Shared Headers Segment is pageable and should be associated with a *temporary* page set, which is to be used as the paging area. Note, however, that the Shared Headers pages are shared among all of the processes of a single arobject, which are executing on a particular node.

Shared Normal Segments:

---

[8] Abstract data type instances can be distributed throughout the nodes of the computer system, although a single instance will always be completely contained within a single node. The indirect invocation of an operation on an abstract data type instance involves first using the information in the Shared Headers Segment to determine which node contains the instance in question. If the instance is on the local node, the operation can be invoked directly. However, if the instance is on a remote node, a form of remote procedure call (RPC) is used in order to invoke the operation on that remote node. See Section RPCSEC for more details on the use of the RPC facility to implement distributed private abstract data types.

There can be multiple Shared Normal Segments within the Shared Region, one for each abstract data type instance which contains "normal" shared data. These segments can be located almost anywhere within the Shared Region, above the Shared Text Segment and below the Shared Headers Segment. However, successive segments will normally be allocated immediately above the Shared Text Segment and any other existing Shared Normal, Shared Permanent, and Shared Atomic Segments. The lifetime of a Shared Normal Segment may be shorter than the lifetime of the address space, since abstract data type instances can be created and destroyed dynamically. Shared Normal Segments permit both read and write access while in user mode, since these segments contain the normal shared variables which define the current states of the abstract data type instances. In addition to being dynamically created and destroyed, Shared Normal Segments can be expanded (grown), to support the dynamic allocation of shared normal variables. Note that since these segments grow upward while the Shared Headers Segment grows downward, the free area near the top of the Shared Region (immediately below the Shared Headers Segment) will be allocated from opposite ends, reducing the chances of conflict. Shared Normal Segments are pageable and should be associated with *temporary* page sets, which are to be used as the paging areas.[9] Note that Shared Normal pages, like Shared Headers pages, are shared among all of the processes of a single arobject, which are executing on a particular node.

Shared Permanent Segments:

Shared Permanent Segments are almost identical to Shared Normal Segments, except that they contain the "permanent" shared variables which define the current states of the abstract data type instances. Also, each Shared Permanent Segment is associated with a *permanent* page set, which is used as both its paging and permanent storage area.[10]

Shared Atomic Segments:

Shared Atomic Segments are identical to Shared Normal and Shared Permanent Segments in most respects, except that they contain the "atomic" shared variables which define the current states of the abstract data type instances. Because of this, each Shared Atomic Segment is associated with an

---

[9] It is possible to use a single temporary page set as the paging area for all Shared Normal Segments, as well as the Shared Headers Segment. However, the use of separate page sets allows the individual paging areas to be located on the same nodes as their corresponding abstract data type instances, increasing efficiency. In addition, it makes it easier to "free" Shared Normal Segments, and to move them around within the Shared Region. (This latter operation may be required in order to avoid conflicts when expanding existing segments.)

[10] Again, it is possible to use a single permanent page set to hold all of the Shared Permanent Segments, but the same considerations as for Shared Normal Segments make the use of separate page sets a bit more attractive.

atomic page set, which is used as both its paging and atomic (permanent) storage area.[11] In order to determine which portions (variables) within Shared Atomic pages have been modified by various transactions, direct writing to the Shared Atomic Segments is not permitted while in user mode (these segments are read-only in user mode). Instead, the special *AtomicCopy* primitive must be used in order to modify any atomic variables. As each modification is made to a Shared Atomic page, the *AtomicCopy* primitive also records the modification in the associated atomic page set. This permits the modification to later be committed or aborted under control of the Transaction Management Subsystem. Note that since each modification is immediately written to the atomic page set, there will never be any need for the Time-Driven Virtual Memory Subsystem to page-out Shared Atomic pages. Also, page-in operations only require the Shared Atomic page to be read from the atomic page set, since the Atomic Page Set Manager will ensure that the page, as read, will reflect all outstanding (uncommitted) modifications.[12] For more details on the handling of atomic objects and transactions, see Section 3.5.

Address Space Management Primitives

The base kernel provides primitives for creating and destroying address spaces, for allocating, freeing, growing, and moving segments within an address space, and for accessing and modifying the information associated with each virtual page within an address space (mapping, flags, and time of last access).[13] The actual address space management primitives provided by the base kernel are the following:

---

[11] In this case too it is possible, though slightly less attractive, to use a single atomic page set to hold all of the Shared Atomic Segments.

[12] The consistency of the atomic data values accessed and modified by a transaction is not guaranteed by these facilities, unless the correct locking protocols are followed. It is assumed that the appropriate read and write locks are obtained for each atomic data item that is to be accessed or modified, respectively. Also note that transaction aborts require notifying the Atomic Page Set Manager of the abort, and also undoing all of the aborted modifications in all of the affected Shared Atomic pages. This latter operation is best accomplished by simply flagging all of the affected pages as "invalid", so that the next attempt to access them will result in a page-in operation. Since the modifications associated with the aborted transaction have been removed from the atomic page set, the page-in operation will read the properly modified contents of the page.

[13] No facilities are provided for accessing or modifying the protection codes governing access to the various parts of an address space. These protection codes are set automatically whenever the address space is created, and whenever new segments are allocated. There should never be any need to modify them explicitly.

```
val = ASCreate(asid [, nvp-shared])
val = ASDestroy(asid)
val = ASActivate(asid)

vpa = ASAllocate(asid, seg-type, nvp, gpsid [, desired-vpa])
val = ASFree(asid, vpa)
val = ASExpand(asid, vpa, nvp)
val = ASMove(asid, vpa, desired-vpa)

ste = ASGetSTE(asid, vpa)
ste = ASNextSTE(asid, vpa)

val = ASSetMap(asid, vpa, ppa [, nvp])
ppa = ASGetMap(asid, vpa)
val = ASSetFlags(asid, vpa, flags [, nvp])
flags = ASGetFlags(asid, vpa)
val = ASSetTime(asid, vpa, time [, nvp])
time = ASGetTime(asid, vpa)
val = ASSetPTE(asid, vpa, pte [, nvp])
pte = ASGetPTE(asid, vpa)

(gpsid, pnum) = ASGetGPSID(asid, vpa)
(asid. vpa, ppa) = ASFindShared(gpsid [, pnum])
```

BOOLEAN val TRUE if the specified operation is completed successfully; otherwise FALSE:

VIRT-PAGE-ADDRESS vpa, desired-vpa
A virtual page address within an address space.

SEG-TABLE-ENTRY ste
A descriptor for a segment within an address space. It includes the segment type (*seg-type*), location (*vpa*), size (*nvp*), and associated page set (*gpsid*).

PHYS-PAGE-ADDRESS ppa
The physical page address in primary memory, to which a virtual page address is mapped.

PTE-FLAGS flags The set of (BOOLEAN) flags associated with a particular page in an address space: USED, MODIFIED, VALID, EXISTS, and COPIED.

VIRT-TIME time The (approximate) virtual (CPU) time at which a particular page in an address space was last accessed.

PAGE-TABLE-ENTRY pte
A descriptor for a particular page in an address space. It includes the corresponding physical page address (*ppa*), *flags*, and last access time (*time*).

GPSID gpsid — Global Page Set ID, which identifies the page set associated with a segment. It includes the ID of the logical disk containing the page set, the page set type (TEMPORARY, PERMANENT, or ATOMIC), and the unique ID of this page set within the logical disk.

INT pnum — A page number within a page set, corresponding to a virtual page within an address space segment.

ASID asid — Address Space ID, which identifies the address space to be operated upon. The corresponding process ID can be easily obtained from an ASID, and vice versa.

INT nvp-shared — The number of virtual pages (size) of the Shared Region.

SEGMENT-TYPE seg-type — The type of the address space segment to be allocated: USER-STACK, USER-TEXT, USER-DATA, USER-HEAP, SHARED-TEXT, SHARED-HEADERS, SHARED-NORMAL, SHARED-PERMANENT, or SHARED-ATOMIC.

INT nvp — The number of virtual pages involved in the operation.

On Error: — Error conditions are indicated by the use of special return values. The details concerning the precise nature of an error condition are provided in the Kernel Error Block.

The *ASCreate* primitive creates a new address space, corresponding to a newly created process.[14] The address space ID (*asid*), which is closely related to the process ID (allowing easy translation back and forth), must be specified as part of the operation. This *asid* will be used to identify the address space in all subsequent operations. The size of the Shared Region (*nvp-shared*) must also be specified if this is the first address space belonging to the corresponding arobject to be created on this node. Otherwise it is optional.[15] Since there is only a fixed, maximum amount of buffer space available for storing address space definitions, the *ASCreate* primitive can fail if too many address spaces have already been defined.[16] The *ASDestroy* primitive can be used to destroy a previously defined address space.

---

[14] It can also be used to *recreate* an address space for a process which had been completely "swapped out", but will soon be required to run again. See Section 3.10 for more details about process swapping.

[15] Since the sizes of the Kernel and Kernel Interface Regions are fixed, specifying the size of the Shared Region will also determine the size of the Private Region (there are no other regions in an address space). Also, if another address space from the same arobject already exists on this node, the size of the Shared Region is already known (all address spaces from the same arobject have identical Shared Regions). The arobject ID can be easily determined from the *asid* (or its associated process ID).

[16] One remedy for this is to swap out and then destroy one of the already existing address spaces..

ASActivate is used when switching processes. It makes the specified address space (*asid*) the currently active one, i.e. it switches the processor to that address space. Note that since the Kernel Region is identical for all address spaces, only the Private and Shared Regions are actually affected by this switch. The processor continues to execute the same code within the kernel as it was prior to switching address spaces. Depending upon the architecture of the system's memory management hardware, activating an address space may require the explicit loading of many registers within the Memory Management Unit (MMU). The management of these MMU registers is solely the responsibility of the address space management routines, especially *ASActivate*. Of course, MMU register management is simplified considerably if the MMU itself handles the loading of its mapping registers, in the manner of a cache.

ASAllocate allocates a new segment within an existing address space. The type of segment (*seg-type*) determines many of its characteristics. In most cases only one segment of a particular type is permitted within an address space. However, multiple SHARED-NORMAL, SHARED-PERMANENT, and SHARED-ATOMIC segments are allowed. At the time a segment is allocated, its initial size (*nvp*) and associated page set (*gpsid*) must be specified.[17] In most cases the location of a new segment is either fixed or can be determined automatically, assuming the various segment types are allocated in the proper order.[18] However, the exact location for a new segment can be specified (*desired-vpa*), whenever necessary.[19] *ASAllocate* returns the location (*vpa*) of the newly allocated segment. For all segments except USER-STACK and SHARED-HEADERS, this is the location of the first (lowest address) page in the segment. For USER-STACK and SHARED-HEADERS, the returned location is the last page in the segment. The returned *vpa* will be used to identify the segment in subsequent operations. *ASAllocate* will fail, returning BAD-VPA, if any conflicts are detected, such as attempting to allocate an already allocated segment type, or overlapping an existing segment.

ASFree deletes an entire segment, indicated by *vpa*, within the specified address space (*asid*). Only SHARED-NORMAL, SHARED-PERMANENT, and SHARED-ATOMIC segments can be freed. [20] *ASExpand* increases the size of an existing segment by the specified number of pages (*nvp*). It will

---

[17] This implies that the associated page set must already exist.

[18] The only restrictions on the ordering of segment allocations are that USER-HEAP must follow USER-DATA, which in turn must follow USER-TEXT, and SHARED-NORMAL, SHARED-PERMANENT, or SHARED-ATOMIC segments must be allocated after the SHARED-TEXT segment.

[19] This should only be necessary when constructing the Shared Region on a new node, so that it matches the Shared Region for an arobject that already exists on other nodes.

[20] This happens as a result of destroying abstract data type instances.

fail if there is insufficient space for the segment to grow by the amount indicated. *ASMove* changes the location of an existing segment from *vpa* to *desired-vpa*. Only SHARED-NORMAL, SHARED-PERMANENT, and SHARED-ATOMIC segments can be moved.[21] *ASMove* will fail if the new location for the segment would overlap another existing segment.

ASGetSTE returns a descriptor for the segment which contains the specified virtual page (*vpa*). This descriptor indicates the segment's type (*seg-type*), location (*vpa*), size (*nvp*), and associated page set (*gpsid*). BAD-STE is returned if the specified page is not within one of the existing Private Region or Shared Region segments.[22] *ASNextSTE* is similar to ASGetSTE, except that it returns a descriptor for the next (higher address) segment which follows, but does not contain, the specified virtual page. Specifying *vpa = 0* will cause the descriptor for the first (lowest address) segment in the Shared Region (SHARED-TEXT) to be returned, or the descriptor for USER-TEXT to be returned if there is no Shared Region. BAD-STE will be returned if the specified page is within or beyond the last segment of the Private Region (USER-STACK). *ASNextSTE* is useful for scanning through all of the segments (and pages) which constitute an address space.

ASSetMap sets the virtual to physical mapping for virtual page *vpa*, to physical page *ppa*. Optionally, a range of *nvp* virtual pages, beginning with *vpa*, can be mapped to contiguous physical pages, beginning with *ppa*. *ASGetMap* returns the physical page address (*ppa*) to which the specified virtual page (*vpa*) is mapped. BAD-PPA is returned if the mapping has not been previously defined using *ASSetMap* (or *ASSetPTE*). *ASSetFlags* sets all of the (BOOLEAN) flags associated with the specified virtual page (*vpa*). Optionally, the flags for a range of *nvp* virtual pages, beginning with *vpa*, can all be set to the same values (*flags*). The available flags are:

- USED: The virtual page has been accessed. This flag helps determine which pages belong to the working set of a process (See Section 3.10).
- MODIFIED: The virtual page has been modified. This flag indicates that the page must be written to the associated page set before the physical page frame can be reused.
- VALID: The physical page address to which the virtual page is mapped is valid, i.e. the page is in primary memory.
- EXISTS: The virtual page exists in the associated page set, i.e. the page has been written some time in the past. This flag helps avoid page-in operations when a newly allocated

---

[21] Moving of segments can be a useful way to recover from *ASExpand* failures.

[22] For our purposes here, the Kernel Stack Segment is not considered a part of the Private Region. Only the "USER" segments are.

virtual page is first accessed.[23]

- COPIED: The virtual page (within the User Data Segment) has been "copied" to the temporary page set, i.e. the temporary page set contains a newer version of the page than the permanent (initial data) page set. This flag only applies to the User Data Segment, and it is used to indicate which page set is to be used when paging-in the virtual page.[24]

ASGetFlags returns the set of flags associated with the specified virtual page (*vpa*).

ASSetTime sets the time of last access for the specified virtual page (*vpa*) to *time*. The *time* value is in virtual (CPU) time units. Optionally, the last access time for a range of *nvp* virtual pages, beginning with *vpa*, can all be set to the same value of *time*. *ASGetTime* returns the last access time for the specified virtual page. NEVER is returned if the page has never been accessed. *ASSetPTE* is equivalent to *ASSetMap*, *ASSetFlags*, and *ASSetTime* combined. It sets all three items of the virtual page descriptor(s) (*ppa*, *flags*, and *time*) to the specified values (*pte*). Similarly, *ASGetPTE* is equivalent to *ASGetMap*, *ASGetFlags*, and *ASGetTime* combined. *ASSetPTE* and *ASGetPTE* are provided as a convenience, for use when entire page descriptors must be modified or retrieved.

ASGetGPSID returns the global page set ID (*gpsid*) and page number (*pnum*) associated with the specified virtual page. This is the page set ID and page number to be used when paging-in or paging-out this virtual page.[25]BAD-GPSID and BAD-PNUM are returned if the specified page is not contained within any of the existing segments in the Private Region or Shared Region. *ASFindShared* searches for the specified page number (*pnum*) from the given page set (*gpsid*), to see if it is already resident in primary memory, and in active use within one of the existing address spaces.[26]If *pnum* is not specified, the search is for any active page from the given page set. If the search is successful, the address space ID (*asid*) and virtual page address (*vpa*) of the first encountered matching entry is returned, along with the corresponding physical page address (*ppa*). Otherwise BAD-ASID, BAD-VPA, and BAD-PPA are returned. *ASFindShared* aids in the handling of "shareable" segments, such

---

[23]*It isn't incorrect to page in a nonexistent page.* It would simply be read as all zeros (see Section 3.7). However, the EXISTS flag helps avoid the overhead of the (unnecessary) page-in operation. Whether a nonexistent page is initialized to zero on first access or simply left undefined depends upon the type of segment it belongs to. User Stack Segment pages can be left undefined, but pages in most other segments should be initialized to zero.

[24]The COPIED flag is essentially another name for the EXISTS flag, as it applies to the User Data Segment. Each User Data Segment page is known to exist in the permanent (initial data) page set. The only question is whether a newer version also exists in the temporary page set.

[25]*ASGetGPSID* takes the COPIED flag into account when determining the page set to be used for pages in the User Data Segment.

[26]This involves searching through the existing address spaces for any segments having *gpsid* as the corresponding page set. The virtual page descriptor for the page corresponding to *pnum* is then checked to see if it is VALID.

as the User Text Segments and all of the Shared Region segments. In particular, it can help determine if page-in or page-out operations are actually required.

Address Space Management Data Structures

The information describing the existing address spaces is contained within kernel data structures called *address space descriptors*. These descriptors are linked together in groups according to the arobjects to which the address spaces belong, as shown in Figure 3-4. This aids in the handling of the Shared Regions of address spaces, especially their construction and modification, since it allows all of the related address spaces to be updated "simultaneously".

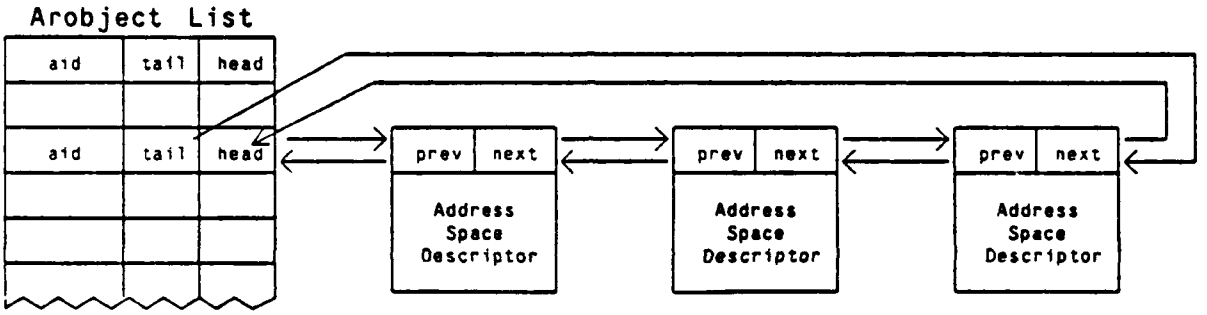


Figure 3-4: Arobject Address Space Lists

The structure of each address space descriptor is illustrated in Figure 3-5. It basically consists of a list of *Segment Descriptors*, which describe each of the segments contained within the address space. Note that since there can be varying numbers of Shared Normal, Shared Permanent, and Shared Atomic Segments, each of these types has its own (sub)list of segment descriptors. Any segment type which is not present in an address space would be indicated by a page count of zero (*nvp = 0*). Each segment descriptor includes a pointer (*pta*) to the *page table*, which describes the state of the individual pages of the segment. Note that each address space within an arobject will actually have its own set of page tables, even for the Shared Region. This is because the Shared Region can be accessed and used in very different ways by the different processes of an arobject, and it is important to determine the virtual memory "working sets" on a per process (per address space) basis (see Section 3.10).

The other main data structure used in the management of address spaces is the Shared Page Set List, illustrated in Figure 3-6. The sole purpose of this structure is to improve the efficiency of the

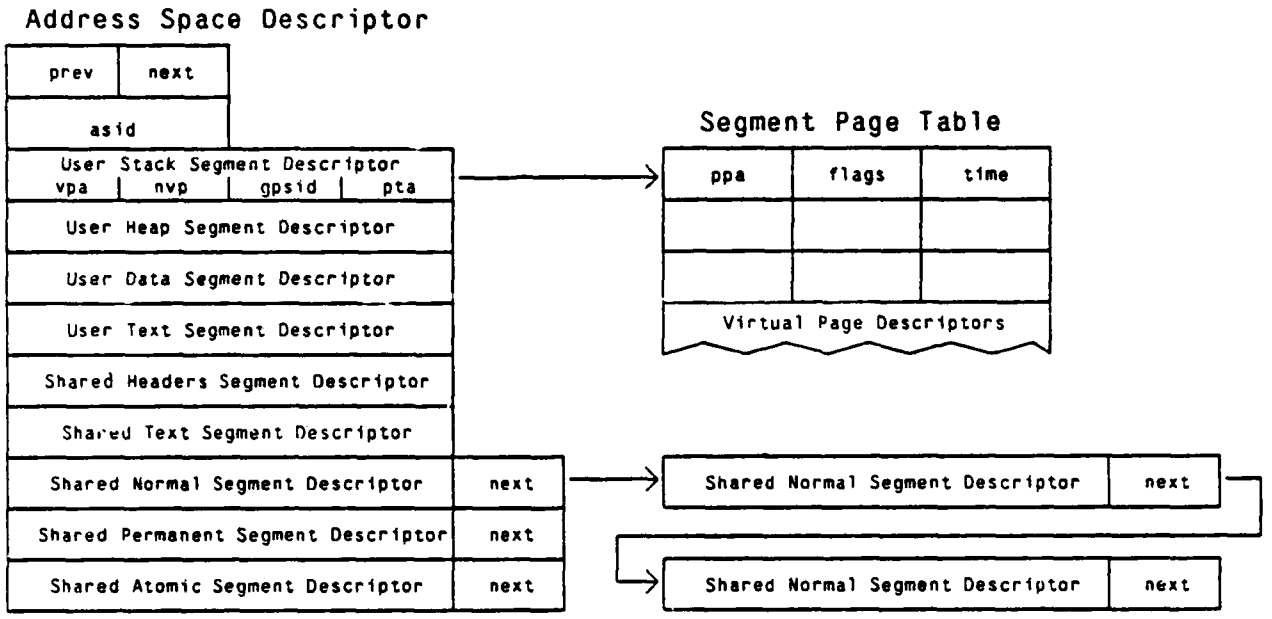


Figure 3-5: Address Space Descriptor

ASFindShared primitive.[28] Before paging-in or paging-out a potentially shared page, i.e. one from the User Text Segment or any segment within the Shared Region, the Time-Driven Virtual Memory Subsystem must first check (using *ASFindShared*) to see if the required page is already in main memory and in use by some other process. If so, the paging operation can be avoided. Given the global page set ID (*gpsid*) for the potential paging operation, *ASFindShared* will look for that *gpsid* in the Shared Page Set List, and then check each associated Segment Page Table to see if the page in question is ever listed as "VALID". Thus, the Shared Page Set List contains an entry for every page set corresponding to a User Text Segment or Shared Region Segment, in any existing address space. Associated with each entry is a list of all the segments (indicated by their address space IDs, virtual page addresses, and page table pointers), which share the use of that page set.

[28] Since the Shared Page Set List could be constructed solely from the contents of the Address Space Descriptors, its use is not strictly required. However, it greatly increases the efficiency of searching for shared pages.

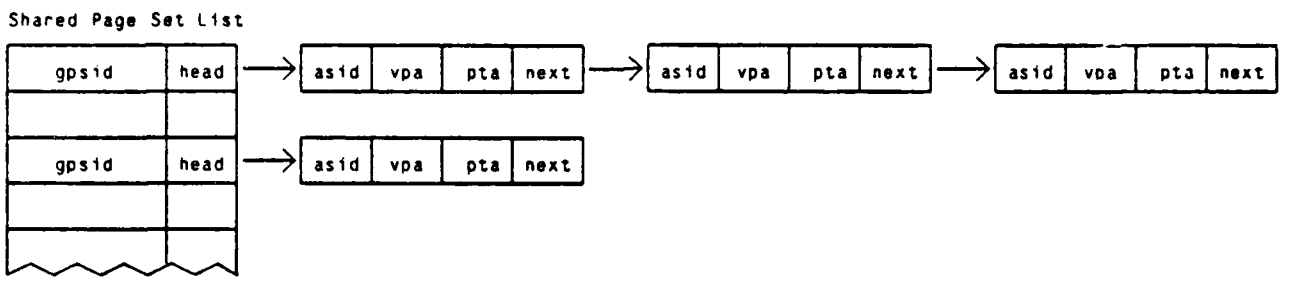


Figure 3-6: Shared Page Set List

3.2.2.6 Synchronization Management

The low-level synchronization mechanisms for arobjects and for basic I/O handling functions are provided at the base kernel. These primitives are designed as a local synchronization mechanism, so remote invocation is not supported at this level. All higher-level synchronization must be performed by using the communication primitives.

The following primitives are provided at the base kernel:

evtcnt = **Sigsend**(event-var)
evtcnt = **Sigrec**(event-var, timeout)
evtcnt = **Sigrecall**(event-var, timeout)
evtcnt = **Sigabort**(event-var, abort-code)

EVTCNT evtcnt The counter value of the specified event variable.

EVENT-VAR event-var
The event variable consists of a waiting queue of client processes and an event counter.

TIMEOUT timeout The timeout value should indicate the maximum execution time for this signal primitive including the waiting time.

ABORTCODE abort-code
An integer value which indicates an abort code.

The *Sigsend* and *Sigrec* primitives are basically similar to the V- and P-operations of an integer semaphore. However, the *Sigrec* primitive will be timed out if the correspoiding signal (e.g., a hardware interrupt) is not generated. A *Sigrecall* primitive is similar to *Sigrec* and used for receiving

all stored event signals. A *Sigabort* primitive can be used to unblock the waiting process with an error condition.

3.3 Arobject/Process Management Subsystem

The Arobject/Process Management Subsystem is responsible for providing arobject/process management facilities in ArchOS. The subsystem consists of a Arobject/Process Manager and its worker processes on each node. The Arobject/Process Manager provides a system-wide facility in corporation with the base kernel. The workers are provided to perform actual work or decision making among cooperating Arobject/Process Managers in the system.

The Arobject/Process Manager primary responsible the following operations:

- Creation and destruction of an arobject and process
- Freezing and Unfreezing of an arobject's or process's activities
- Binding and unbinding of reference names for arobjects and processes
- Allocation and deallocation of private data objects
- Internal access mechanisms to fetch and store any data objects for an arobject/process.
- Recovery management for atomic arobjects

It should be noted that many functions can be invoked locally by the base kernel. This subsystem is necessary to provide the service for remote invocations.

3.3.1 Arobject/Process Management

An Arobject/Process Manager exists on each node and manages a fixed number of workers. Every worker can perform the following service functions:

- Coordinate with the other Arobject/Process Manager to perform the best assignment decision for creation of a new arobject/process instance.
- Propergate a new reference name to the other Arobject/Process Manager.

The basic components of the arobject/process subsystem is shown in Figure 3-7.

When a new instance of an arobject or process is created, a system-wide unique identifier called an *arobject id (aid)* or *process id (pid)* is created and is also guaranteed to be unique over the lifetime of the arobject or process. For a new instance of an arobject, an *arobject descriptor* is created and

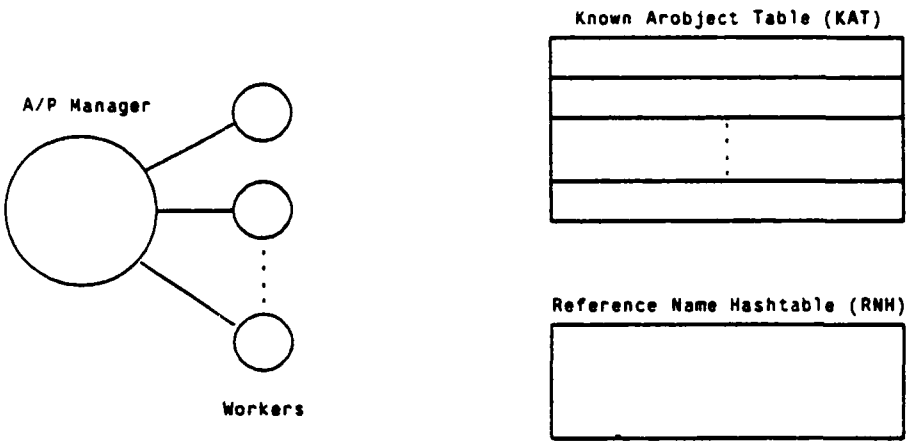


Figure 3-7: Components of the Arobject/Process Subsystem

registered in the *known arobject table* in the kernel. For a new process instance, a *process descriptor* is also allocated and linked to its arobject descriptor.

The arobject descriptor contains the following information to control its arobject components:

- Arobject id (aid):
  An aid is a fixed-length descriptor consisting of *current node id*, *birth node id*, and *local unique id*. Aid is used to identify a destination arobject where a request operation will be invoked, so the current node id and birth node id are included for reducing the searching process and migration process.

- Parent aid:
  Its parent's aid.

- Arobject status:
  Status of the arobject.

- Freeze/Unfreeze event variable:
  An event variable to control .

- A set of reference pointers to private object descriptors:
  A private object descriptor contains the data associated with the arobject's private object, namely "processes", "shared abstract data types", "statistics data object", "message queue", etc.

For instance, suppose that arobject A has two processes (INITIAL and P_1) and one shared private object (SO_1) and a new instance of A is created at node X. After the INITIAL process is started, it creates process P_1 at node Y. Then, the relationship among the major kernel objects are shown in Figure 3-8.

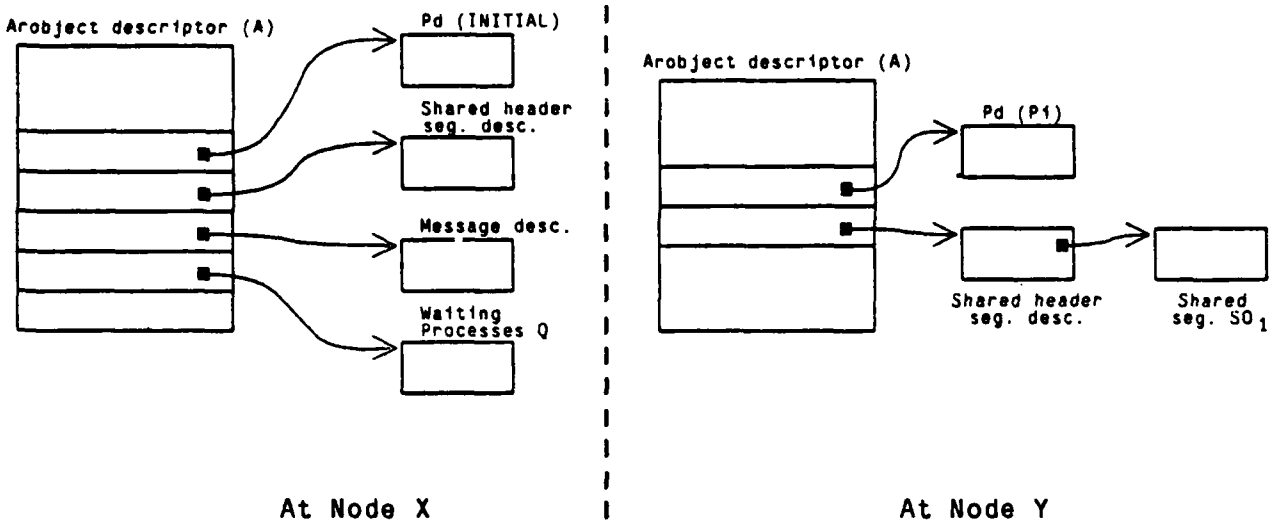


Figure 3-8: An example of Arobject Descriptors

At node X, an arobject descriptor is allocated for an instance of arobject A. A process descriptor for INITIAL and a shared object descriptor for the header segment and for SO_1 are linked to the arobject descriptor. At node Y, there are also an arobject descriptor for A and a process descriptor for P_1 which is linked to the arobject descriptor. Although there is no shared private object created at node Y, a shared object descriptor is also allocated for the header segment and linked to the arobject descriptor.

It should be noted that the message queue of arobject A is only allocated on node X. Thus, when P_1 attempts to accept a request message, a remote operation is invoked to fetch a message from the message queue at node X. If there is no acceptable request message in the queue, P_1 will be blocked and placed in the waiting processes' queue.

3.3.1.1 Arobject/Process Assignment Policy

When creation of a new instance of an arobject or process is requested at a non-specific node, the arobject/process manager selects the best node according to the current "arobject/process assignment policy".

The assignment policy, like other user definable policies, can be set by a system designer. In order to reduce system overhead, the policy definition module for the assignment policy is placed in Arobject/Process Manager.

Without creating a new policy definition module, the Arobject/Process Manager can provide the following assignment policies.

- First Fit (FF): The first A/P manager which replies the creation request will be selected.
- Random Fit (RF): A A/P manager from the random selection will be used.
- Best Fit (BF): One of the best matched A/P manager will be selected.
- Best Effort Fit(BEF):

3.3.1.2 Life cycle of Arobject/Process

The life cycle of an arobject depends upon whether the arobject is an atomic or nonatomic. When an instance of an arobject is instantiated, the arobject instance becomes *active*. If an arobject is atomic, it can be *inactivated* and remained on stable storage. On the other hand, if an arobject is nonatomic, it cannot be inactivated and it must be *dead* when it is killed. Atomic and nonatomic arobject can be *frozen* for monitoring or debugging purpose.

The life cycle of process is similar to the arobject's one. When an instance of a process is created, it becomes *ready* and runnable. Once the time-driven scheduler deciedes to run a process, it becomes *running* and the process may block due to I/O waiting or scheduler's preemption. Like arobject, a process will be *dead* when it is killed and it can be also *frozen*. The life time of a process instance depends on the life time of its arobject. When the arobject is killed, all of its processes will be also killed by the system. Thus, there will be no frozen processes left in its arobject.

The life cycle of an arobject and process is depicted in Figure 3-9.

3.3.1.3 ArchOS primitives

The following ArchOS primitives are supported for arobject/process management for a client.

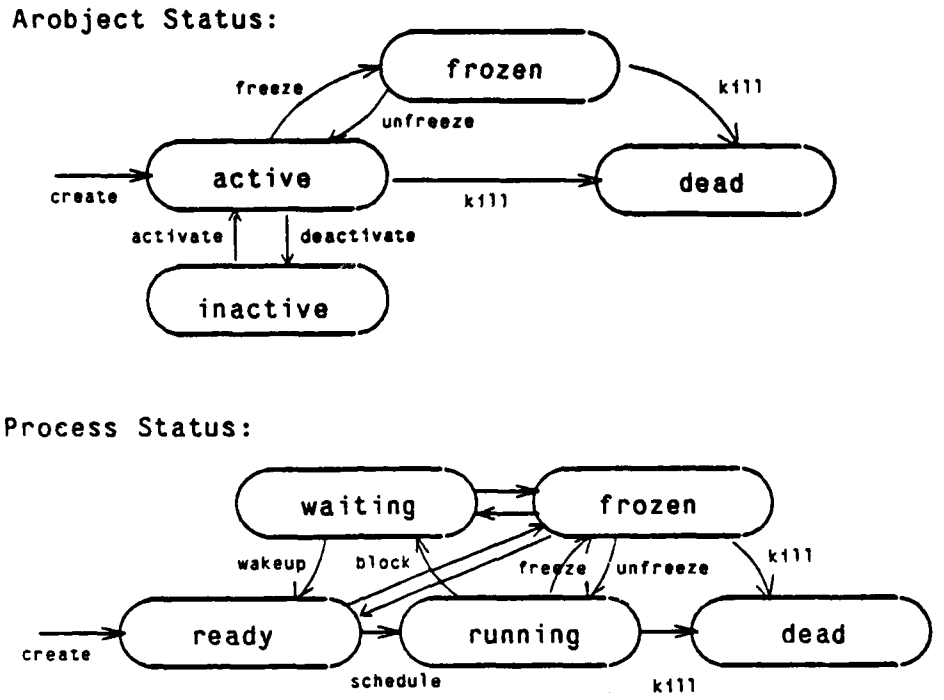


Figure 3-9: Life Cycle of an Arobject and Process

arobject-id = **CreateArojbect**(arobj-name [, init-msg] [, node-id])
process-id = **CreateProcess**(process-name [, init-msg] [, node-id])
val = **KillArobject**(aid)
val = **KillProcess**(pid)
val = **GlobalKillArobject**(arobj-id, kill-options)
val = **GlobalKillProcess**(pid, options)

aid = **SelfAid**()
pid = **SelfPid**()
paid = **ParentAid**(aid)
ppid = **ParentPid**(pid)

val = **SetErrorStack**(errblock, blocksize)
val = **FreezeArobject**(arobj-id, options)
val = **UnfreezeArobject**(arobj-id, options)
val = **FreezeProcess**(pid, options)
val = **UnfreezeProcess**(pid, options)

fval = **FetchArobjectStatus**(arobj-id, dataobj-id, buffer, size)
sval = **StoreArobject**(arobj-id, dataobj-id, buffer, size)
fval = **FetchProcessStatus**(pid, dataobj-id, buffer, size)
sval = **StoreProcess**(pid, dataobj-id, buffer, size)

BOOLEAN val — TRUE if the primitive was executed properly; otherwise FALSE.

NODE-ID node-id — The node id indicates the actual node which will be stopped.

AID arobj-id — The unique arobject id of an arobject instance.

PID pid — The process id of the target process.

GKILL-OP kill-options
The options indicate various control options. For example, it can indicate whether the caller stops every time after killing a single process or not.

FREEZE-OPT options
The options indicate various selectable flags such as a timeout freeze/unfreeze flag.

INT fval — The actual number of bytes which were fetched.

INT sval — The actual number of bytes which were stored.

DATAOBJ-ID dataobj-id The dataobj-id indicates the private object or system control status of the target arobject/process.

BUFFER *buffer — A pointer to the buffer area for storing the returned data object value.

INT size — The size indicates the buffer size in bytes.

A *CreateArobject* primitive creates a new instance of an arobject at an arbitrary node or a specified node. Similarly, a *CreateProcess* primitive creates a new instance of a process in the arobject. The selection of a node is made automatically by ArchOS unless overridden by the *Create* operation. Upon arobject creation, the arobject's INITIAL process is automatically dispatched. An optional set of parameters can be passed to the INITIAL process when the arobject is instantiated by using an initial message (i.e., "init-msg").

The *KillArobject* and *KillProcess* primitives remove a process and arobject instance respectively. An arobject may be killed only by one of its own processes (suicide allowed, no murder). In order to kill another arobject, the target arobject must have an appropriate operation defined within its specification so it can kill itself. A process can be killed only by a process which exists in the same arobject instance.

The *SelfAid* primitive returns the requestor's arobject id and the *ParentAid* primitive returns the

parent's arobject id of the specified arobject. The *SelfPid* primitive returns the process id (pid) of the requestor and the *ParentPid* primitive returns the parent's pid of the the specified process.

A *SetErrorBlock* primitive sets an error block in a process's address space. A user error block consists of a head pointer and a circular queue. The head pointer contains a pointer to an entry which contains the latest error information in the circular queue. After the execution of this primitive, a client can access the detailed error information from the specified error block. A *FreezeArobject* primitive stops the execution of an arobject (i.e., all of its processes), and a *FreezeProcess* primitives halts a specific process for inspection. An *UnfreezeArobject* and *UnfreezeProcess* primitive resumes a suspended arobject and process respectively. While a process is in a *frozen* state, many of the factors used for making scheduling decisions can be selectively ignored. For instance, a timeout value will be ignored by specifying a proper flag in the *Freeze* primitive.

A *Fetch* primitive inspects the status of a running or frozen arobject or process in terms of a set of frozen values of private data objects. The specific state of the arobject or process will be selected by a data object id. The state includes not only the status of private variables, but also includes process control information.

A *GlobalKill* primitive can destroy an arbitrary arobject or process in the system.

3.3.1.4 Creation and Destruction of Arobject/Process

A client can create a new instance of arobject at any node in the system by issuing the CreateArobject primitive. The first argument, *arobj-name*, contains three values: The first value contains a file name for the image of the shared region and the second value points to a file for the image of the private region of its INITIAL process. The last section contains a table of entry points for light-weight processes. Note that the light-weight process is available for kernel arobjects.

For instance, arobject name A may contains {"/usr/test/a.arobj", "/usr/test /p0.proc", nill}. When a client executes 'CreateArobject(A, msg, Y)", at first the base kernel determines whether the target node is local or remote. Since this is a remote invocation request, it looks up the A/P manager's *reference name hash-table* and determines the destination A/P manager's AID. Then, the client sets up a request message for the remote A/P manager and issues a proper invocation request.

When the remote Communication manager's NetIn worker receives the request packet, it placed in the target A/P manager's request queue. If one of its workers is already waiting on the incoming invocation request, the request message will be placed directly into the worker's message buffer.

Once the worker executes the CreateArobject primitive, the virtual address space for arobject A is created by reading "/usr/test/a.arobj" and "/usr/test p0.proc" files. A new arobject descriptor is also allocated and its AID is returned to the worker. Then, the worker returns the result message to its A/P manager and the A/P manager forward the result to the caller's A/P manager. Then, the original caller receives the result from the local A/P manager.

The sequence of interaction between two A/P manager is depicted in Figure 3-10.

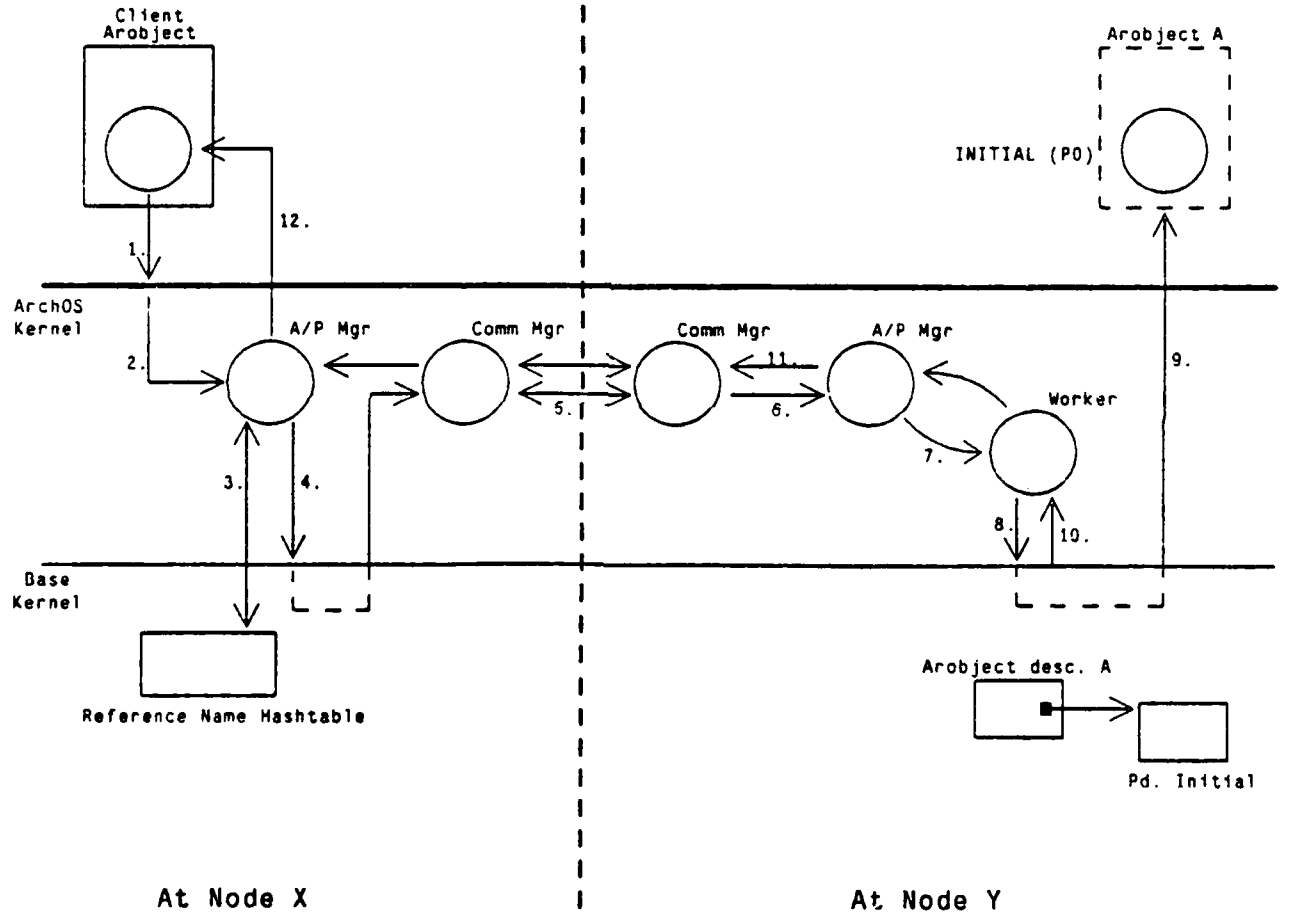


Figure 3-10: Creation and Destruction of an Arobject and Process

3.3.2 Name Management

An arobject/process manager maintains binding information in a hash table. When a reference name is bound to a caller, the caller's arobject/process manager registers its name first. Then, the manager's worker must propagate its entry across the system using one-to-many communication (i.e., using a *RequestAll* primitive). The lifetime of a binding is the same as the lifetime of an arobject or process instance.

To find one or all arobject/process instances from its reference name, the arobject/process manager searches its own hash table and if there is no entry there, then it inquires from the other managers.

The following ArchOS primitives are supported for name management for a client:

---

val = **BindArobjectName**(aid, arobj-refname)
val = **BindProcessName**(pid, process-refname)
val = **UnbindProcessName**(pid, process-refname)
val = **UnbindArobjectName**(aid, arobj-refname)

aid = **FindAid**(arobj-refname [, preference])
pid = **FindPid**(process-refname [, preference])
aid-list = **FindAllAid**(arobj-refname [, preference])
pid-list = **FindAllPid**(process-refname [, preference])

AID aid — The arobject id.

PID pid — The process id.

AID-LIST aid-list — The list of corresponding aid's.

PID-LIST pid-list — The list of corresponding pid's.

AROBJ-REFNAME arobj-refname
The reference name of related arobject(s).

PROCESS-REFNAME process-refname
The reference name of related process(es).

PREFERENCE preference
The preference can specify a search domain such as "INTERNAL", "EXTERNAL", "LOCAL", "REMOTE", "INTERNAL-LOCAL", "INTERNAL-REMOTE", "EXTERNAL-LOCAL", "EXTERNAL-REMOTE".

The *BindArobjectName* and *BindProcessName* primitives bind the requested instance of an arobject

or process to a reference name. This binding allows an arobject or a process to have more than one reference name, or a single reference name can be bound multiple arobject or process instances. To cancel the current binding, a process must use the appropriate *Unbind* primitive.

A *FindID* primitive returns the unique id (i.e., aid or pid) of the given arobject or process in a specific search domain A search domain can be specified with respect to all of the internal arobjects, external arobjects, a local node, a remote node, or a reasonable combination of among four. If more than one instance uses the same reference name, the unique id of any one of them will be returned. A *FindAllID* primitive, on the other hand, returns all of the aid's and pid's which correspond to the given reference name.

3.3.3 Private Object Management

Private object management allows a client process to allocate and deallocate an instance of a private abstract data type at any node. An Arobject/Process manager maintains a list of private object descriptors under its arobject descriptor to keep the data associated with it.

Since an object type can be one of *Normal*, *Permanent*, and *Atomic*, the private object manager must coordinate with a page set manager (see Section 3.7) to allocate a proper type of page set to create a new instantiation of the abstract data type.

If a remote allocation is requested, the creation of a new private abstract data type must be coordinated between the destination's Arobject/Process manager and the one with the INITIAL process. Since every process share all private segments and should have a uniform view of local and remote instances, the shared header segment must be updated by the Arobject/Process manager at the INITIAL process's node. When a proper update is done, updated part of the shared header segment is propagated to the other Arobject/Process manager.

---

object-ptr = **AllocateObject**(type-name, object-type, parameters, [, node-id])
val = **FreeObject**(object-ptr)
val = **FlushPermanent**(object-ptr, size)

OBJECT-PTR object-ptr
A pointer to the allocated private data object.

OBJECT-TYPE object-type
The object-type indicates the name of a private abstract data type.

BOOLEAN val TRUE if the object was released successful; otherwise FALSE.

NODE-ID node-id Node identification. An actual node may be designated, or a node selection

	criterion may be designated (e.g., the current node, any node except the current node, any node, or a specific node).
INT size	The number of bytes which must be flushed into permanent storage.

An *AllocateObject* primitive allocates an instance of a private abstract data type at any node and a *FreeObject* primitive deallocates the specified instance. A *FlushPermanent* primitive blocks the caller until the specified data object is saved in non-volatile storage.

3.3.4 Recovery Management

The arobject/process manager is responsible for restarting *atomic arobjects* which have at least one private atomic data object in the event of node failures. Since all private atomic objects are kept on a corresponding atomic page set, the arobject/process manager will coordinate with the page set subsystem to resume the crashed atomic arobject by recreating its initial process with the pre-crash image of the private atomic data objects and permanent data objects, if any.

It should be noted that it is still an application designer's responsibility to determine what recovery action must take place based on its atomic and permanent data objects.

3.4 Communication Subsystem

The communication subsystem provides intra- and inter-node message communication mechanisms to support a system-wide, location- independent operation invocation for cooperating arobjects. An invocation request can be initiated by referring to the destination arobject's id and the operation name from anywhere in the system. Although a single invocation is initiated through the arobject id, multiple invocation can be initiated by referring to a reference name of arobjects which offer the same service. In this case, a calling arobject may continue to perform its activity and receive one or more results by using a *GetReply* primitive.

The communication subsystem also interacts with the transaction subsystem to coordinate the necessary transaction management. For instance, a *Request* primitive is treated as an elementary transaction consisting of three steps: a sending part of the *request*, an invocation linking part, and a receiving part of the *request*. Then, the linking part links the requestee's *Accept*, computation, and *Reply*. Since all message activities in this transaction are defined as compound transactions and the invocation linking part is defined as an elementary transaction, it is possible for the requestee's computation to be a nested elementary or compound transaction of the top-level transaction.

3.4.1 Message Header and Body

A message consists of header and body parts. The header part is not writable from a client and only the body part can be set by a client.

The message header contains the following data:

- Transaction id
- Requestor's arobject id
- Destination arobject id
- Destination operation name
- Number of arguments
- Size of message in bytes

It should be noted that even though message typing is not supported at runtime, type checking of messages between requestor and requestee can be done solely at compile time. Since message communication preserves message boundaries, it does not offer a "stream-oriented" communication interface at this level.

3.4.2 Message Queue

There are two types of message queues associated with arobject and process instances. A *request-queue* is allocated for an instance of an arobject. When a new arobject instance is created, the arobject/process manager creates a request-queue associated with its arobject descriptor. That is, the body of the message queue is kept in the kernel at the running node of the initial process. A *reply-queue* is allocated for an instance of a process. When a process issues an invocation request to an arobject, its results will be placed in the reply-queue.

In both message queues, each entry is represented by a *message descriptor* and can be checked without accepting the message itself.

3.4.3 Communication Manager

When an arobject invocation request is issued at a caller's site, the communication manager sets up a proper header part for the message packet. The message then is sent to the destination arobject. If the destination arobject is in a remote node, the remote invocation protocol is used and the communication manager becomes the monitor for the protocol. Similarly, when a process accesses a

shared private data at a remote node, a remote procedure call is used and the communication manager executes its protocol.

3.4.3.1 Components of the Communicatin Manager

Each Communication Manager works with a pair of network I/O workers called, "NetIn" and "NetOut" and a group of "Stub" workers.

NetIn and NetOut workers are resposible to receive and transmit a message packet between two nodes. A Stub worker is used to perform a remote invocation request for a shared privated data object. When a remote invocation is received by the Communication Manager, it assigns the acutual work to the stub worker. Then, the stub worker calls the target procedure with the arguments and returns the result packet to the caller. A sequence of remote invocation is described in Section .

3.4.3.2 ArchOS primitives

The following ArchOS primitives are supported for communication management for a client.

```
trans-id = Request(arobj-id, opr, msg, reply-msg)
trans-id = RequestSingle(arobj-id, opr, msg)
trans-id = RequestAll(arobject-name, opr, msg)
pid = GetReply(trans-id, reply-msg)

(trans-id, requestor, opr) = AcceptAny(acc-opr, msg)
(trans-id, opr) = Accept(requestor, acc-opr, msg)
trans-id = Reply(pid, req-trans-id, reply-msg)

ptr-mds = CheckMessageQ(qtype, selector, selector-id)

val = CaptureCommArobject(arobj-id, commtype, requestor, req-opr)
val = CaptureCommProcess(pid, req-opr)
val = WatchCommArobject(arobj-id, commtype, requestor, req-opr)
val = WatchCommProcess(pid, req-opr)
```

TRANSACTION-ID trans-id
: The transaction id of the transaction on whose behalf the request is being made.

AID arobj-id
: The unique id of the receiving arobject.

OPE-SELECTOR opr
: The name of the operation to be performed.

MESSAGE *msg
: A pointer to the message which contains the parameters of the operation to be performed. The message to the destination arobject must not contain any pointers (i.e., call-by-value semantics must be used).

REPLY-MSG *reply-msg
A pointer to the reply message.

AROBJ-REFNAME arobj-refname
The reference name of the receiving arobject(s).

OPE-SELECTOR acc-opr, req-opr
The name of operation to be performed. The "opr" parameter can be a specific operation name or "ANYOPR".

TRANSACTION-ID req-trans-id
The transaction id of the transaction on whose behalf the request is made.

MSG-DESCRIPTORS *prt-mds
Pointer to a list of the message descriptors selected by the specified selection criteria.

MSG-Q qtype, commtype
This indicates either "request-" or "reply-" message queue.

BOOLEAN val TRUE if the primitive was executed properly; otherwise FALSE.

AID requestor The aid of the communicating arobject.

The *Request* primitive provides remote procedure call semantics in which the requesting process invokes an operation by sending a message and blocks until the receiving arobject returns a reply message. Identically, if the receiver arobject is the same arobject, a local operation will be invoked.

The *RequestSingle* and *RequestAll* primitives can send a request message and proceed without waiting for a reply message. The *RequestSingle* primitive provides nonblocking one-to-one communication and, the *RequestAll* primitive supports one-to-many communication. The requesting process may thus invoke an operation on more than one instance of an arobject or process with one request. To receive all of the replies, the *GetReply* primitive may be repeated until a reply with a null body is received.

The *GetReply* primitive receives a reply message which has the specific transaction id generated by the proceding *RequestAll* primitive. If the specific reply message is not available, then the caller will be blocked until the message becomes available.

The process responsible for an arobject operation receives a message using the *Accept* primitive. Using the selection criteria specified, the operating system selects an eligible message from the

arobject's input queue and returns it. The process operates on the message, responding with a reply when processing has been completed. If no suitable message is in the request message queue, then the caller will block until such a message becomes available.

The *AcceptAny* primitive can receive a message from any arobject instance with any operation (i.e., "ANYOPR") or a specified operation request. The primitive can return the requestor's transaction id, specified operator, and requestor's aid. The *Accept* primitive can receive a message from a specific requestor arobject and returns the requestor's transaction id and the requested operator.

The CheckMessageQ primitive examines the current status of an incoming message queue without blocking the caller process. The primitive must specify a message queue type, either "request-queue" or "reply-queue". The request-queue queues all of the non-accepted request messages and is allocated for each arobject instance. The reply-queue maintains all of the non-read reply messages and is assigned to every process instance. A message can be selected based on the sender's arobject id, operation name, and/or transaction id. If more than one argument is given, only messages which satisfy all of the conditions will be returned. If no corresponding message exists in a specified message queue, a *"NULL-POINTER" will be returned.*

3.4.4 Remote Invocation Protocol

A remote invocation protocol is used to control the invocation of an operation on a remote arobject.

In a simple remote invocation case, two packets will be exchanged between two nodes: one is a *request* packet and the other is *reply* packet. A simplified version of protocol sequence is shown in Figure 3-11.

When a remote request primitive is issued by a client, the base kernel pass its request to the communication manager. Then, the communication manager places a request into an outgoing packet queue and gives it to a "NetOut" worker. The NetOut worker simply initiates the actual output activity over the net. Note that the NetOut worker can be replaced by a simple procedure call within the communicaiton manager if the context switching is significantly large.

When a remote site receives the request packet, the "NetIn" worker of the communication manager places the packet in its incoming packet queue and notify it to the communication manager. If the destination arobject is ready to accept the request, the communication manager moves the actual message to the destination arobject's receiving buffer. Otherwise, the communication manager places the message in the destination arobject's request queue.

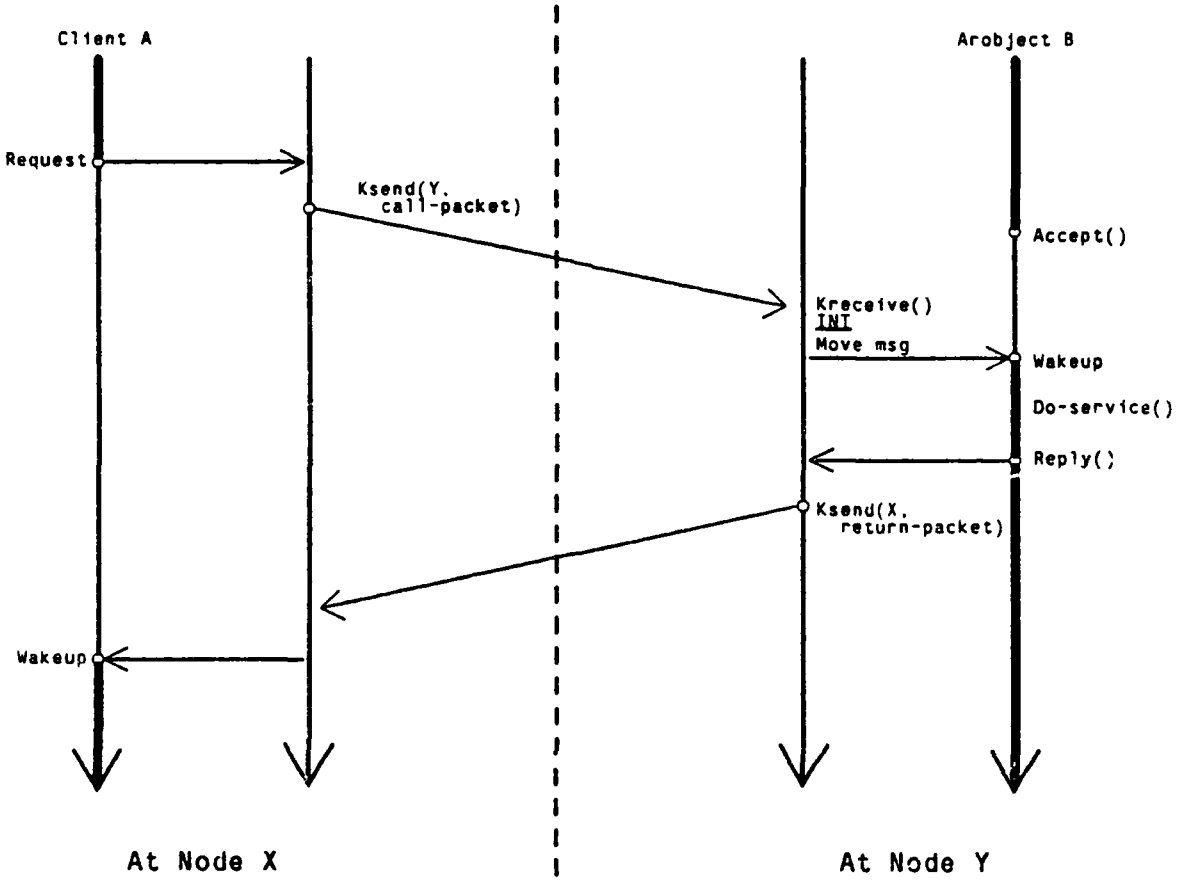


Figure 3-11: Remote Invocation Protocol

3.4.5 RPC for Shared Private Objects

@Lable(rpcsec)

When a client invoke an operation on a shared private object exists on a remote node, the communication manager's *stub* worker performs the actual invocation and returns the result to the caller.

At first, the private object invocation request to a is checked by looking at the shared object table in the shared header segment. If the target object is in a remote node, a remote invocation request, *InvokePrivate* will be sent from the communication manager.

When the remote "NetIn" worker receives the packet, it checks availabiility of the stub workers. If

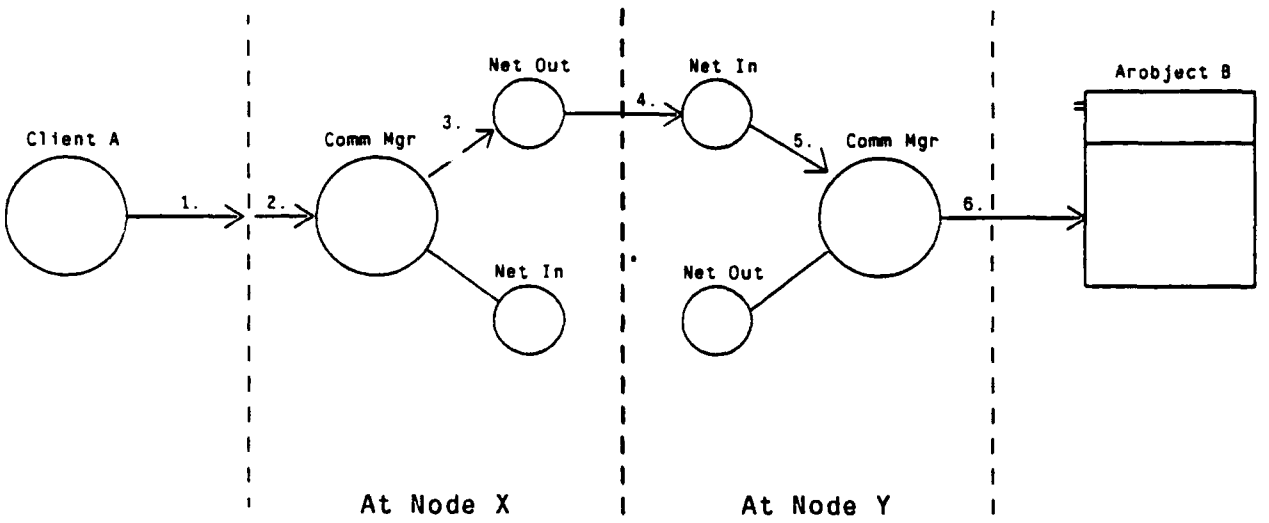


Figure 3-12: Interaction Sequence for a Remote Invocation Request

one of stub workers is idle, it assigns the actual invocation request to the stub worker. When the stub worker completes the invocation request, it returns the result message to the communication manager. Then, the communication manager forwards the result packet to the caller.

The basic sequence within in a remote communication manager is shown in Figure 3-13.

3.5 Transaction Subsystem

The transaction subsystem manages two types of transactions: compound transactions and elementary transactions. The transaction subsystem allows a client to nest a compound transaction or an elementary transaction in any combination. By using the nested elementary transactions, a client can use a traditional "nested transactions" mechanism [Moss 81]. A compound transaction can be used on a "compensatable" atomic arobject. [Sha 85, Tokuda 85] Thus, the transaction subsystem must provide the uniform control mechanisms to perform a "commit" operation for both types and an "undo" operation for the elementary transactions and a "compensation" operation for the compound transactions.

It should be noted that the current transaction subsystem does not provide any mechanisms which support a "cooperating" transaction.

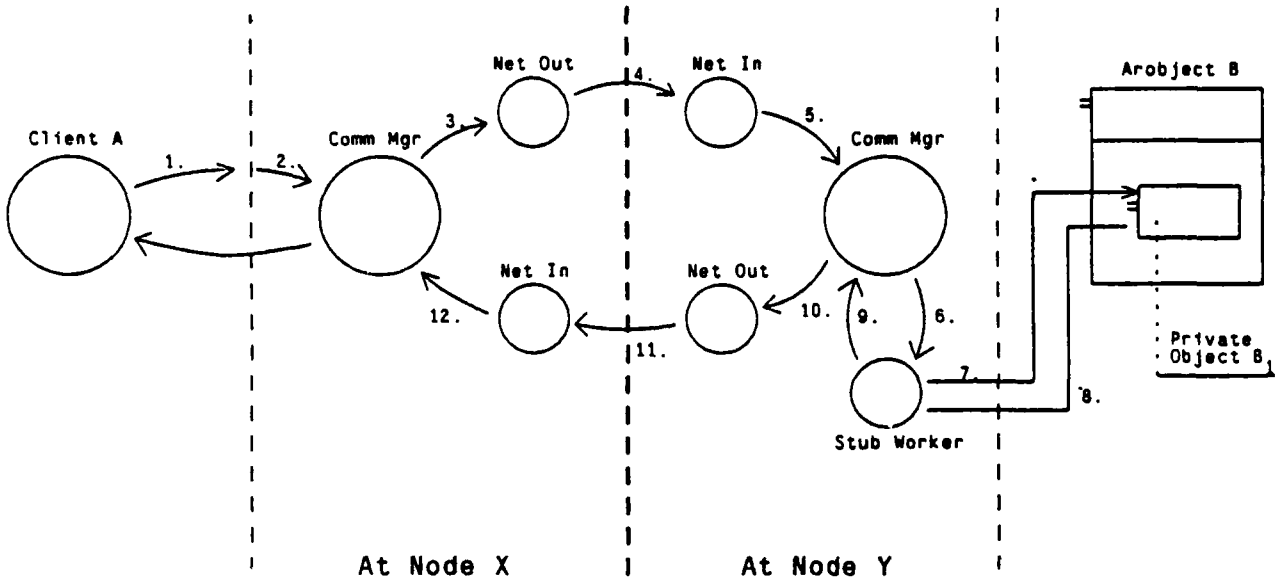


Figure 3-13: Remote Procedure Call Sequence at a Remote Communication Manager

3.5.1 Transaction Types, Scopes, and Tree

- **Transaction Types:**
  A transaction type indicates whether the current transaction is the elementary or compound transaction and the top-level or not.

- **Transaction Scope:**
  A transaction scope is created when a new transaction construct is used in a process. Within this scope, a client can access atomic objects as if these computational steps were executed alone.

- **Transaction Tree:**
  A transaction tree is a structure that can be used to trace the dynamic behavior of a set of transactions. The transaction tree also provides information related to the commit protocol and lock propagation among the nested transactions.

- **Transaction Id:**
  The transaction subsystem gets a unique transaction id from the base kernel when a new transaction is initiated and guaranteed to be unique over the lifetime of the transaction. A transaction id is a fixed-length descriptor consisting of a sequence of *parent's node id*, *current node id*, and *local unique id*.

3.5.2 Transaction Management

The transaction manager is responsible for maintaining the consistency of the atomic arobjects even in the case of a node failure. The major activity of the transaction manager is the "BeginTransaction", "Commit" and "Abort" transaction processing.

To commit a transaction, the transaction manager coordinates with the related page set subsystem and performs atomic update for all atomic objects the transaction has visited. The actual coordination is performed based on the 3-phase commit protocol [Bernstein 83] to improve its reliability.

To abort a transaction, the transaction manager must also coordinate with the page set subsystem, but it performs "undo" for elementary transactions and initiates the necessary "compensation operations" for all pre-committed compound transactions.

3.5.2.1 Components of the Transaction Manager

Each Transaction Manager works with a group of workers, called "Coordinator" and "Subordinator". A "Coordinator" is assigned to keep track the activity of a top-level transaction. When the top-level transaction is committed, the "Coordinator" worker is responsible to perform 3-phase commit protocol among the related subtransactions. A "Suordinator" is used to keep track the activity of a nested transaction.

When a new transaction is created, a transaction descriptor is created in the Transaction Manager and is used to maintain the current status of the transaction and relationship to its child transactions.

3.5.2.2 ArchOS primitives

The following ArchOS primitives are supported for transaction management for a client.

```
tid = BeginTransaction(trantype, timeout)
sval = CommitTransaction(tid)
sval = AbortTransaction(tid)
sval = AbortIncompleteTransaction(req-tid)

tid = SelfTid()
ptid = ParentTid(tid)
trantype = TransactionType(tid)
val = IsCommitted(tid)
val = IsAborted(tid)
```

TIME timeout The timeout value indicates the maximum lifetime of this elementary transaction.

TRANTYPE trantype

	The type of the given transaction, such as "CT", "ET", "Nested CT", or "Nested ET".
TID tid	The id of the specific transaction.
TID ptid	The parent's tid of the given transaction "tid".
INT sval	returns 1 if the requested action was performed on the specified transaction successfully; otherwise returns error staus code.
BOOLEAN val	TRUE if the requested predicate is hold; otherwise returns FALSE.

The *BeginTransaction* primitive creates a transaction descriptor for the requested transaction. The *CommitTransaction* primitive commits the specified transaction.

The *AbortTransaction* primitive aborts the specified transaction and all of its child transactions within the same transaction tree. If the transaction that invokes the *AbortTransaction* primitive does not belong to same the transaction tree as the transaction which is to be aborted, a client cannot abort that transaction. This primitive executes all of the necessary "undo" or "compensate" actions, based on the transaction type, and breaks the current transaction scope. After completion of these actions, the status of all affected atomic objects will be consistent and returned to either "identical" to or "a member of the equivalence class" of their initial (pre-execution) states. The *AbortIncompleteTransaction* primitive also aborts all of the outstanding incomplete transactions which had been initiated by an outstanding *RequestSingle* or *RequestAll* primitive. In other words, all of the nested transactions which belong to the specified request transaction but have not yet completed (committed) will be aborted.

The *SelfTid* primitive returns the id of the current transaction and the *ParentTid* primitive returns the p ent transaction id of the given transaction id. The *TransactionType* primitive returns the type of the given transaction (a compound or elementary) and also indicates the transaction level. A *IsCommitted* primitive checks whether the given transaction is already committed or not. A *IsAborted* primitive checks whether the given transaction is already aborted or not.

3.5.3 Three-Phase Commit Protocol

When a top-level transaction is created, its node's transaction manager becomes a primary coordinator to perform the 3-phase commit protocol and the other transaction managers where at least one nested transaction was executed become subordinators. These subordinator transaction managers are also used for backup of the primary manager.

The current 3-phase commit protocol [Bernstein 83] can be summarized as follows:

1. The transaction manager, say TM_i, which is responsible to a top-level transaction, T_{top}, invokes "prepare" operations at all visited node's transaction managers by using a RequestAll primitive. Then, it waits for all transaction managers to acknowledge the "prepare" operation.

2. TM_i invokes "precommit" operations on all the other transaction managers which involve T_{top}. When the other transaction managers initiate the "precommit" operation, they add a new entry, T_{top}, to its copy of "commit list" of T_{top}. Then, they wait for all acknowledgements for "precommit" to come back.

3. TM_i perform "commit" operations at all related transaction managers.

3.5.4 Compensation Action Management

A transaction manager must initiate a compensation action whenever a compound transaction is aborted and must guarantee that the effects of all committed actions are cancelled out. In general, the ordering relation among the compensation operations is sensitive to satisfy local and global "equivalence relations", thus the transaction manager must maintain a clear ordering rule. In the current algorithm, we take the reverse ordering of the "committed" sequence of operations.

The book keeping is performed by using a "compensation log" at each arobject. When an arobject invokes a compensatable operation on another arobject as a nested transaction, a "compensation log" record which consists of the current transaction id (i.e., caller's transaction id), arobject name, operation name, parameters, the compensation operation's name, necessary parameters for the compensation operation, will be added on the caller's log. When the caller's transaction is aborted, the system first checks the "cancellation relation" between the current operation and the previous operations by getting the information from the arobject and then the system can simply determine the necessary compensation operations by looking at the compensation log record from the end to the beginning and invoke them.

3.5.5 Lock Management

Proper lock management is necessary to provide a consistent view of atomic data object across the transaction tree.

When a nested elementary transaction commits, all of the locks that it held are passed to the transaction in which it is nested (its parent in the transaction tree); when a nested compound transaction commits, all of its locks are released. The rules that determine which locks a transaction may obtain are more involved. Two transactions are said to be *unrelated* if: (1) they are not contained

in a single transaction tree, or (2) they are in a single transaction tree and are concurrently executing siblings or descendants of concurrently executing siblings, or (3) they are in a single transaction tree in which one is an ancestor of the other and either the descendant is a compound transaction or there is a compound transaction on the path connecting the two transaction nodes in the corresponding transaction tree. (Note that according to this definition, a compound child transaction is always unrelated to its parent transaction.)

Two unrelated transactions may compete for locks, and they may hold locks with compatible lock modes for a single data object at any given time. However, if they request incompatible lock modes for a single data object, then one of the competitors will obtain a lock and the other will block until it can receive the desired lock, or it will return to the requestor with an appropriate status indication.

Two transactions are said to be *related* if they are contained in a single transaction tree where one is the descendant of the other, the descendant is an elementary transaction, and there are no compound transactions in the path connecting their respective nodes in the transaction tree. The lock compatibility rules for related transactions are different than those for unrelated transactions. In this case, the descendant transaction can obtain any lock mode for any lock held by a related ancestor in the transaction tree, including incompatible lock modes that would not be allowed if the transactions were unrelated. (Of course, the descendant transaction will have to compete with all of the unrelated transactions in the system to successfully obtain the requested lock with the desired mode.)

The following ArchOS primitives are supported for lock management for a client:

---

newlock-id = **CreateLock**([parent-lockid])
val = **DeleteLock**(lockid)

sval = **SetLock**(lock-type, lockid, lock-mode)
tval = **TestandSetLock**(lock-type, lockid, lock-mode)
tval = **TestLock**(lock-type, lockid, lock-mode)
rval = **ReleaseLock**(lock-type, lockid, lock-mode)

INT sval — 1 if the specified lock is set; 0 if the lock is not set. A negative value will be returned if an error occured.

INT tval — 1 if the specified lock is being held; 0 if the lock is not being held. A negative value will be returned if an error occured.

INT rval — 1 if the specified lock is released; 0 if the lock is not released. A negative value will be returned if an error occured.

LOCK-TYPE lock-type
The lock type can be either "TREE" or "DISCRETE".

LOCK-ID lockid The lockid indicates the unique id of a lock.

LOCK-MODE lock-mode
The lock mode can be "READ", "WRITE", etc.

The *SetLock* primitive sets a "tree-type" or "discrete-type" lock on arbitrary objects by specifying a lock key and its mode. If a requested lock is being held, the caller will block until it is released. The *TestandSetLock* primitive also tries to set a lock, however, it will return a "FALSE" if the lock is being held. If the request lock is a tree-lock type, then the *SetLock* and *TestandSetLock* primitives may also fail due to the violation of the tree-lock convention (See Section 3.5.5).

The *TestLock* primitive checks the availability of a specified lock with a lock mode. In the case of a tree lock, it also checks whether the locking would be legal in the corresponding lock tree. The *ReleaseLock* primitive can release the lock on an object which was gained by the *SetLock* or *TestandSetLock* primitive explicitly.

3.5.6 Recovery Management

Since automatic recovery of application programs is not an easy task for the transaction subsystem, the subsystem guarantees only the consistency of atomic arobjects and provides a handle to distinguish the pre-crash and post-crash situation. Each arobject designer must define a proper action for recovery and let the "INITIAL" process handle a detailed recover sequence.

In ArchOS, each arobject has an atomic variable, called the "restart-counter" which becomes incremented whenever its host machine restarts. By using the restart counter, the INITIAL process can examine whether the program is running before a crash or not.

To clean up and retrieve the necessary atomic arobjects, the transaction subsystem must coordinate with the page set manager. In particular, the atomic page set manager can recover the nessary page set for a given atomic arobject.

3.6 File Management Subsystem

The ArchOS File Management Subsystem provides a system-wide, location independent file access service. It provides three types of files: *normal*, *permanent*, and *atomic*. *Normal* files are temporary, and not recoverable following a crash. *Permanent* files are saved on disk, and most closely resemble files on other systems. *Atomic* files are guaranteed to be failure atomic under "soft and clean" failures [Bernstein 83]. All three types of files are client level arobjects when active (i.e. open). When inactive, permanent and atomic files are saved using page sets (See Section 3.7).

The implementation of files as regular client level arobjects achieves two main advantages. First, the full power of the ArchOS transaction facilities is available, permitting the arbitrary nesting of atomic file operations within other nested transactions. Second, the file arobjects are treated in the same manner as other arobjects by the Time-driven Scheduler and the Time-driven Virtual Memory Manager. This ensures that critical files (data) will be available (scheduled and paged in) when they are to be accessed by critical processes, thus reducing the number of missed deadlines.

The File Subsystem uses logical, location independent names for files. Hence, files are not constrained to reside on particular disk volumes. They can be dynamically moved to other disks by the system, if desired, for improved global disk usage. The file name space is flat (not hierarchical), but facilities are provided to allow many of the benefits of hierarchical directories. File names are expected to be relatively long strings, composed of several shorter strings (called *components*), separated by the special character '/' (slash). An initial substring of a name (upto a slash) is known as a *prefix*.

The system-wide file system directory is partitioned, where each partition is saved in a known location (refer Figure 3-14). For example, the part of the directory containing all file names with a particular prefix is saved at one logical disk. This style of implementation was dictated by efficiency considerations, such as in creating and locating files. Facilities are provided for moving parts of the directory from one disk to another, but this is expected to occur infrequently.

The functionality of the File Management Subsystem is provided by four separate but co-operating entities (refer Figure 3-15).

1. Each client arobject is linked with a standard file system interface. This interface consists of a library of routines, which hide the implementation details of the other subsystems within the File Management Subsystem.

2. Each open (active) file is represented by an instance of a user level file type arobject.

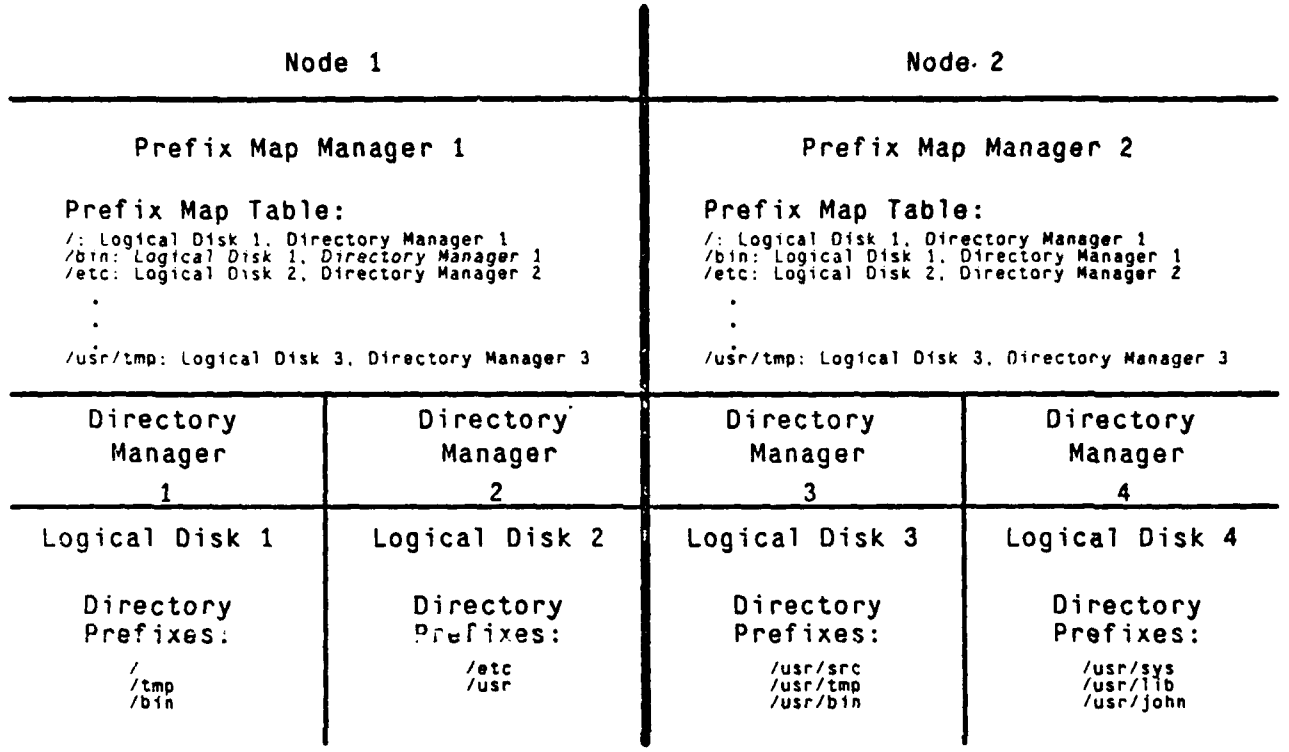


Figure 3-14: The Partitioned File Subsystem Directory Structure

3. Each node in the system contains a kernel level Prefix Map Management Arobject, which is responsible for mapping file name prefixes to the logical disks on which their directory entries reside.

4 Each logical disk in the system containing a portion of the directory has an associated kernel level Directory Management Arobject.

The above four entities will be described in some detail in the following sections.

3.6.1 Client Arobject's File System Interface

A client arobject will always interact with the file system through the use of a standard set of library routines. These routines are together referred to as the Client Arobject's File System Interface (or the CA Interface). The primary purpose of the CA Interface is to transparently interact with the three other subsystems of the file system (File Arobjects, Prefix Map Manager, and Directory Manager) in supporting client level functionality. Thus, it hides the details of the interactions with the other parts of the File Subsystem, and provides a cleaner interface than the raw system primitives.

A second aspect of the CA Interface's functionality is that it provides appropriate internal buffering to improve efficiency. For open files, it also keeps track of the sizes of the files.

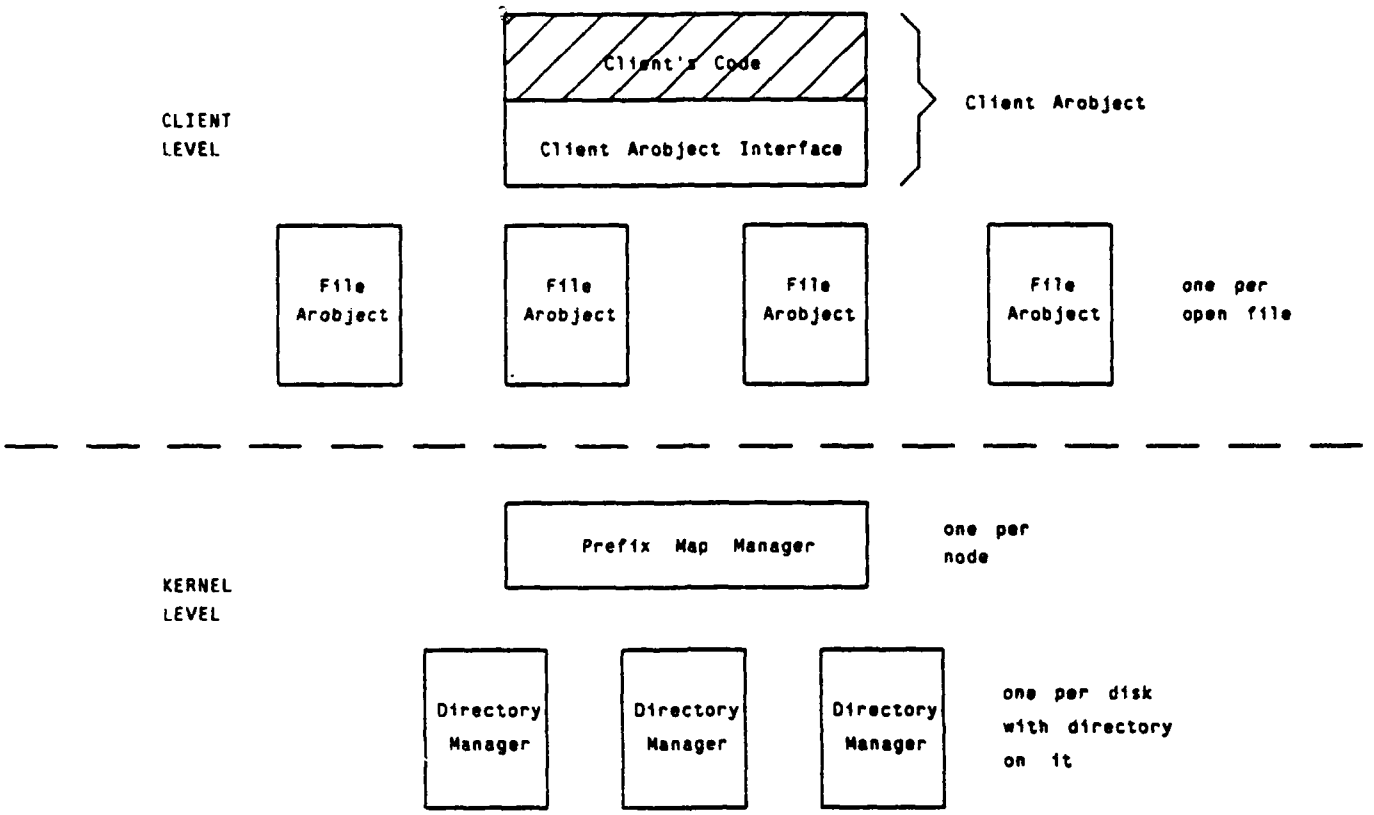


Figure 3-15: Subsystems within the File Management Subsystem

The following list of primitives is intended to illustrate the functionality of the ArchOS standard file I/O library. It is not an exhaustive list. Many other primitives can be added to make file management more convenient for the client.

```
fid = CreateFile(filename, filetype, mode [, node])
val = DeleteFile(filename)
fid = OpenFile(filename, mode)
val = CloseFile(fid)
val = RenameFile(oldname, newname)
val = SyncDirectory(filename)

val = SetPrefix(prefix)
prefix = GetPrefix()

nread = ReadFile(fid, buffer, length)
nwritten = WriteFile(fid, buffer, length)
nwritten = ZeroFile(fid, length)
val = SeekFile(fid, byteoffset, origin)
val = SyncFile(fid)

val = StatusFile(fid, statusbuffer)
val = InfoFile(filename, infobuffer)
```

INT fid — The ID for the file.

BOOLEAN val — TRUE if the specified operation is done successfully; otherwise FALSE.

FILENAME prefix — The filename prefix currently in use, which will be added to any filename tails being specified.

INT nread — The number of bytes actually read.

INT nwritten — The number of bytes actually written.

FILENAME filename — The name of the file for which the specified operation is to be performed.

FILETYPE filetype — The type of the file to be created: NORMAL, PERMANENT, or ATOMIC.

MODE mode — One of four possible modes in which the file can be opened: READ, WRITE, READ-ONLY, and EXCLUSIVE-WRITE.

NODENAME node — The ID of the preferred node on which the file should be created.

FILENAME oldname, newname — The old and new names (respectively) of the file being renamed.

BUFFER *buffer — The address for the data buffer.

INT length — The number of bytes to be read or written.

INT byteoffset The position in number of bytes from the origin.

FILEORIGIN origin Starting position in the file, which can have one of three values: START-OF-FILE, END-OF-FILE, and CURRENT-POSITION.

FSTATUSBUFFER *statusbuffer
The buffer address for returning dynamic file status information.

FINFOBUFFER *infobuffer
The buffer address for returning static file information.

The *CreateFile* primitive creates a file of the specified type (NORMAL, PERMANENT, or ATOMIC) on the preferred computing node. If a node preference is not specified, the file is created on any node. The newly created file is then opened in the specified mode. The *DeleteFile* primitive deletes the specified file. *OpenFile* opens the specified file in one of four modes: READ, WRITE, READ-ONLY, and EXCLUSIVE-WRITE. The *CloseFile* primitive closes the specified file.[29] *RenameFile* changes the name of the file from the old name to the new name specified. The *SyncDirectory* primitive is used for saving any buffered directory information for the specified file on disk.

The *SetPrefix* primitive provides a short hand technique for giving filenames. A file prefix can be specified, which is automatically concatenated with filename tails to form the complete file name. The *GetPrefix* primitive allows the client to obtain the current value of the file prefix.

The *ReadFile* primitive is used for reading the specified number of bytes from a file, starting at the current position. The bytes read are returned in a buffer in the client's address space. Similarly, the *WriteFile* primitive writes the specified number of bytes from the buffer, at the current position in the file. The *ZeroFile* primitive is used for writing the specified number of zero bytes, starting at the current position in the file. The ability to zero a file is useful when truncating files, and creating sparse files. The *SeekFile* primitive allows random access to a particular position in the file. The position can be specified as a byte offset from the start or end of the file, or from the current position in the file. The *SyncFile* primitive flushes the contents of a file from buffers in memory onto disk.

The *StatusFile* primitive is used for obtaining dynamic status information of a file in the buffer provided. Information such as the current mode of file access, and the number of active and outstanding open requests on the file is provided. The *InfoFile* primitive, on the other hand, provides static file information, such as creation date, last modification date, length of file, etc.

---

[29] All open files for a client are registered with the kernel by the CA Interface. Hence, if an arobject with open files is destroyed, the kernel can close all these open files. The occurrence of each of the three primitives *CreateFile*, *OpenFile*, and *CloseFile*, is reported to the kernel.

3.6.2 File Arobjects

Each open file in the system is represented by a client level file arobject. There are three different types of file arobjects, corresponding to the three different types of files: normal, permanent, and atomic. A file arobject maintains the "file buffer", and provides byte level file I/O. The file is mapped onto the virtual address space of the file arobject, and the buffer is maintained automatically by the Virtual Memory Subsystem. The two main functions of a file arobject are: (1) maintaining locks, and thus ensuring consistent concurrent access to the file, and (2) handling *read* and *write* operations on the file. Locks are set on the entire file. When the file is opened, the type of access is specified, and the appropriate lock is set. The lock is released only when the file is closed. In addition to lock management, reading and writing, a few other primitives such as *FASync* and *FAStatus* are also provided.

```
filesize = FAOpen(mode)
val = FAClose()

nbr = FARead(location, nbytes, buffer)
nbw = FAWrite(location, nbytes, buffer)
nbw = FAZero(location, nbytes)

val = FASync()
val = FAStatus(statusbuffer)

val = FARestart(fdm-aid, filesize)
```

INT filesize	The size of the file in bytes.
BOOLEAN val	TRUE if the specified operation is done successfully; otherwise FALSE.
INT nbr	The actual number of bytes read.
INT nbw	The actual number of bytes written.
MODE mode	One of four possible modes in which the file can be opened: READ, WRITE, READ-ONLY, and EXCLUSIVE-WRITE.
INT location	The location (in bytes from the beginning of the file) within the file at which reading or writing starts.
INT nbytes	The number of bytes to be read or written.
BUFFER *buffer	The address for the data buffer.
FASTATUSBUFFER *statusbuffer	The buffer address for returning status information.

AID fdm-aid	The ID of the Directory Management arobject, to be used when closing the file.

The *FAOpen* primitive allows a file to be opened in one of four possible modes: "READ", "WRITE", "READ-ONLY", and "EXCLUSIVE-WRITE". These modes apply to all three types of files, but the behavior of a mode does depend on the type of file. Once the file is opened in a particular mode, the appropriate type of lock is set on it to ensure consistency. The size of the file is returned to the client arobject. The file size, as well as the current position pointer are maintained by the Client Arobject Interface. The *FAClose* primitive closes the file. It also deactivates the file arobject if there are no other outstanding "opens" on the file.

The *FARead* and *FAWrite* operations allow multiple bytes to be read or written at a time. The current position at which reading or writing should start, and the number of bytes to be read or written are provided, along with a pointer to the data buffer[30] The actual number of bytes read or written is returned by the operation. The *FAZero* primitive allows a number of bytes inside of the file to be zeroed out. Thus, it is possible to truncate a file, or maintain a sparse file in this system.

The *FASync* operation flushes the contents of the file buffer onto the disk, so that any buffered information is synchronized with the copy of the file stored on disk. The *FAStatus* primitive returns dynamic file information in the statusbuffer provided. Information is provided about the mode of file access, the lock compatibility of open files, and the number of active and outstanding open requests.

The *FARestart* operation is used for initialization purposes when the file arobject is first created (which can happen when the file is created, or first opened). It is invoked by an instance of the Directory Management arobject, which provides its own ID and the current size of the file as parameters. The ID of the Directory Management arobject is used by the file arobject when the file is closed by any client.

3.6.2.1 Components of a File Arobject

The components of a File Arobject are described below and shown in Figure 3-16. Each file arobject has a single process called the *FA Manager Process*, which receives all requests for operations, carries out the appropriate operations, and returns the results. It operates on three main data structures: (1) the Open Client List (OCL), (2) the Request Client List (RCL), and (3) the file buffer. The *Open Client List* maintains a list of the clients which have successfully opened this file,

[30] If the client is remote with respect to the file arobject, the communication mechanism automatically handles data transfer to the remote buffer.

and the mode of access for each client. Since locking is done on a per file basis, all these clients must have compatible locking on the file. The *Request Client List* is a list of the clients waiting for their open operations to be completed. The information maintained for each request is a triple consisting of the request transaction ID, the ID of the requesting client arobject, and the mode of access requested.

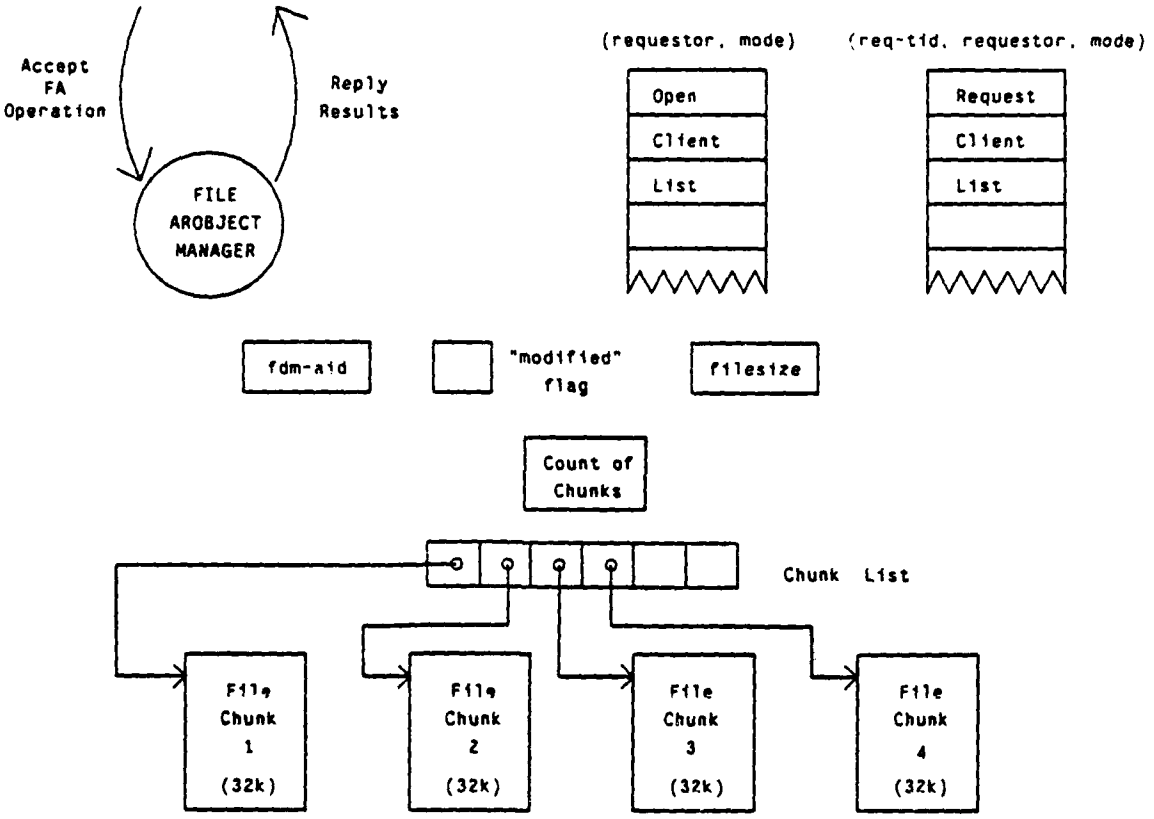


Figure 3-16: Components of a File Arobject

The file buffer is a part of the file arobject's address space. It is maintained in a number of fairly large chunks (32 KBytes per chunk[31]). A list of pointers to these chunks and a count of the number of chunks are also maintained. The advantage of large chunks is that a long list of chunk pointers is not needed, and hence searching for a particular page in the file can be more efficient.

[31] The size of the chunks is determined by the architecture of the SUN workstations, the current target machines for ArchOS implementation.

In addition to the three main data structures, a few other pieces of information are also maintained. Some of these are the ID for the Directory Management arobject, a flag which indicates whether the file has been modified, and a variable which saves the current size of the file[32].

3.6.2.2 Normal and Permanent File Manipulations

Opening of files is allowed to proceed on a FCFS basis[33], but multiple compatible opens are permitted. When a *FAOpen* request is received by the FA Manager process, the Open Client List is first checked to see whether there are other ongoing requests. If the OCL is empty, the incoming request is placed on that list. If the Request Client List has some waiting clients, then the new request is added to that list. If the RCL is empty, and the *mode* of the incoming request is compatible with the modes of the ongoing requests in the OCL, the new request is added to the OCL, but if the mode is incompatible, then it is added to the RCL. Once the incoming request has been added to the OCL, the *FAOpen* is complete, and the client is signalled. If the incoming request is added to the RCL, the *FAOpen* is suspended, pending lock availability.

When the *FAClose* operation is requested, the OCL is first checked for the requesting client, and the corresponding entry is removed. Next, the FDClose() operation is invoked for the appropriate Directory Management (FDM) arobject. The FDM arobject maintains a count of outstanding open requests, and informs the File Arobject whether it should continue, deactivate, or kill itself. If the FA has to continue, it moves all of the compatible entries from the head of the RCL to the OCL; and notifies the corresponding clients. If there are no outstanding open requests, the FA deactivates itself. If the file has been deleted, and the last *FAClose* operation has completed, the FA kills itself.

The reading and writing of normal and permanent files is very straightforward. The system "copy" mechanism carries out the transfer of data between the FA address space and the client's address space. Reading and writing to disk is automatically handled by the virtual memory manager. Each time the *FAWrite* and *FAZero* operations are invoked, the file arobject updates its notion of the current file size.

---

[32]The size of the file saved here is updated whenever *FAWrite* and *FAZero* operations are invoked. This is not the most up-to-date value, since the Client Arobject Interface buffers several read and write operations.

[33]The policy for opening files can possibly be set by the user to better suit the constraints of the real time application.

3.6.2.3 Atomic File Manipulations

The operations on atomic files are quite similar to their counterparts for normal and permanent files. The main difference is that atomicity of the operations has to be guaranteed whenever the file is manipulated. The *FAOpen* and *FAClose* operations are the same as described in the previous section, since these operations only affect the OCL and RCL data structures, and do not touch the atomic file data.

The *FARead* and *FAWrite* operations are implemented as elementary transactions, so that they can be arbitrarily nested inside of other transactions. The *FARead* operation invokes the *Copy* mechanism for copying the data from the file arobject's address space to the client's address space. For the *FAWrite* operation however, the *AtomicCopy* operation is invoked, which updates the file in the main memory, and also propagates the write operation to the Atomic Page Set Subsystem (refer Section 3.7.2).

3.6.3 Prefix Map Management

The directory of the ArchOS File Subsystem is distributed across multiple nodes and multiple disks of the system. Specifically, all directory entries with a particular prefix are on a particular logical disk. Hence, we need to maintain a system wide table which can map the directory fragments corresponding to different prefixes, to the logical disks on which these fragments reside. The main function of the File Prefix Map Management Subsystem (or FPMM subsystem) is to maintain this mapping in a table known as the Prefix Map Table (or PMT). There is one instance of the FPMM subsystem at the kernel level of each node of the distributed system. A copy of the entire system wide Prefix Map Table is maintained by each FPMM instance.

All of the functionality provided by the FPMM subsystem is related to mapping file name prefixes to IDs of appropriate Directory Management Arobjects, which save the directory entries for those prefixes. Once the appropriate Directory Mangement Arobject for a file has been determined, all future operations on that file are performed either by the File Aroject, or by the Directory Mangement Arobject (depending on the operation in question). The FPMM subsystem, on the other hand, provides operations for accessing and maintaining the Prefix Map Table, e.g. at restart, and when disk volumes are mounted and unmounted. The ability to create and delete new directory fragments by adding or removing file prefixes is also provided.

---

fdm-aid = **FPMap**(filename)

val = **FPRestart**()
val = **FPMount**(diskid)
val = **FPUnmount**(diskid)
val = **FPAssign**(prefix, diskid)
val = **FPUnassign**(prefix)

val = **FPInsertTable**(prefix, diskid, fdm-aid)
val = **FPRemoveTable**(prefix)
val = **FPRequestTable**(tablebuffer)

AID fdm-aid — The arobject ID of the Directory Management subsystem.

BOOLEAN val — TRUE if the specified operation is done successfully, otherwise FALSE.

FILENAME filename
The name of the file for which the specified operation is to be performed.

DISKID diskid — The Logical Disk ID of the disk volume being operated on.

FILENAME prefix — The filename prefix being used in the specified operation.

TABLEBUFFER *tablebuffer
The buffer used for returning the contents of the Prefix Map Table.

The *FPMap* primitive returns the arobject ID of the Directory Mangement Subsystem instance which manages the directory fragment for the given file. This primitive will usually be invoked by the Client Arobject Interface, when the first operation (e.g. *CreateFile* or *OpenFile*) is requested by a client. Once the appropriate Directory Management Arobject for a specific file has been determined, any further requests relatd to that file will not be addressed to the FPMM Subsystem.

The main purpose of the *FPRestart* primitive is to reconstruct the Prefix Map Table. This is done by determining which disks are mounted on this node, and by acquiring information from the PMTs residing at other nodes in the system. The *FPMount* primitive is used when a disk volume is mounted. If the new disk has a portion of the directory on it, the Prefix Map Table is modified to reflect this, and an FDM arobject is created to manage that directory. The *FPUnmount* primitive is used when a disk volume is removed. Any entries corresponding to this disk in the Prefix Map Table are removed, and the FDM arobject is informed.

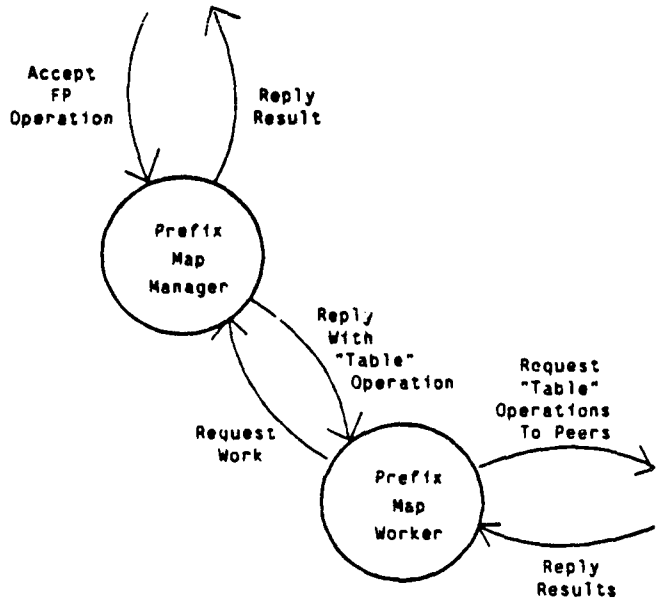
The *FPAssign* and *FPUnassign* primitives are used for adding and removing prefixes to and from the entire file system directory. A new prefix can be added to the specified disk irrespective of whether the disk already has a part of the directory on it or not. When a new prefix is added, modifications are made to the Prefix Map Table and to the disk itself. An FDM arobject for the disk also has to be created, if it does not already exist. The *FPUnassign* primitive removes a prefix from a disk by making modifications to the PMT and to the disk. It also informs the FDM arobject if necessary. This primitive will execute only if the prefix being unassigned does not correspond to any existing files; otherwise an error indication is returned.

Three primitives are provided for obtaining and modifying information from the Prefix Map Table. These primitives will be invoked primarily by FPMM arobjects on other nodes of the system. The *FPInsertTable* primitive inserts a new entry into the PMT, which consists of the directory prefix being added, the ID of the logical disk on which the directory portion exists, and the ID of the FDM arobject responsible for the management of that directory portion. The *FPRemoveTable* primitive removes the specified prefix entry from the PMT. The *FPRequestTable* primitive asks for all the contents of the PMT to be returned in the buffer provided.

3.6.3.1 Components of the Prefix Map Management Subsystem

The FPMM subsystem consists of three components: (1) the Prefix Map Table or the PMT, (2) the Prefix Map Manager process, and (3) the Prefix Map Worker process (refer Figure 3-17). The PMT keeps a copy of the entire system wide mapping between prefixes and directory locations. Its entries are a series of triples consisting of the file name prefix, the ID of the logical disk containing the directory fragment corresponding to this prefix, and the ID of the FDM arobject which manages this directory fragment. It is important to ensure the consistency of the multiple copies of the PMT.

The Prefix Map Manager process is responsible for providing most of the functionality of the FPMM subsystem. It accepts all requests for FPMM operations, and returns the results. The only purpose of the Prefix Map Worker process is to wait on behalf of the Manager process when peer level primitives are invoked by the Manager process. These primitives (FPInsertTable, FPRemoveTable, FPRequestTable) are used for maintaining system wide consistency between the PMT instances. In the absence of the Worker process, deadlocks can arise if multiple peer operations are in progress concurrently.



Prefix Map Table

File Name Prefix	Logical Disk ID	Directory Manager AID

Figure 3-17: Components of the Prefix Map Management Subsystem

3.6.3.2 Directory Mounting and Reassignment

In order to understand the implementation of the primitives *FPMount*, *FPUnmount*, *FPAssign*, and *FPUnassign*, it is important to have some information about the layout of the logical disk volumes. There is a special page (Page 0) on each disk volume which contains information about the contents and location of several important data structures stored on the disk. Hence, by examining Page 0, it is possible to determine whether there are any directory fragments on the disk, and if so, what file prefixes they correspond to.

When a logical disk is mounted on a node, the *FPMount* primitive is invoked on that node to update the PMT if necessary. First, Page 0 of the disk is checked to see whether the disk has a directory fragment on it, and if so, to determine the prefixes which are mapped on it. All these prefixes are then entered in the local copy of the PMT, along with the ID of the logical disk. An FDM arobject is created and initialized to manage the newly mounted disk directory, and its arobject ID is entered in the PMT. This completes the updating of the local PMT. All newly added entries are now sent to all other instances of FPMM by invoking the *FPInsertTable* primitive on peer FPMM arobjects.

When a disk is unmounted, the *FPUnmount* primitive is invoked. If there are any entries in the PMT

which correspond to this logical disk, these entries are removed. The FDM arobject for that disk is cleaned up and destroyed by invoking the *FDUnmount* primitive. Furthermore, peer FPMM arobjects are informed by invoking the *FPRemoveTable* primitive.

The *FPAssign* primitive assigns a new prefix to a particular logical disk. First Page 0 of the disk is checked to see whether it already has a directory on it or not. If there is no directory, a directory root page set is created on that disk, and a pointer to its root page is entered in Page 0, along with the new prefix being added. An FDM arobject also has to be created to manage the directory on this disk. An entry is added to the PMT corresponding to the new prefix, logical disk, and FDM arobject. The addition of a new prefix to a disk which already has a directory fragment on it is much simpler. It only requires a new entry on Page 0 of the disk and in the PMT, since the directory root page set and FDM arobject are already present. In both cases, once the PMT has been updated, the peer FPMMs are informed of the new entry.

The *FPUnassign* primitive removes a directory segment (corresponding to a prefix) from a disk. However, this operation is only allowed when there are no files in the system which correspond to that prefix. First, the prefix is removed from Page 0 of the disk. If, as a result of this removal, there are no more prefixes left on the disk, the directory root page set also has to be freed. If the disk has no directory segment left on it, the FDM arobject for the disk is destroyed by invoking *FDUnmount*. In any event, the entry for the prefix is removed from the PMT, and the peer FPMMs are informed of the removal.

3.6.3.3 Restart

The main purpose of the *FPRestart* primitive is to recreate the Prefix Map Table. To do this, the Manager process first invokes the Worker process to obtain the PMT (with FPRequestTable) from one of the other nodes in the system. In the meantime, the Manager itself checks the Mount Table of its node (see Section 3.7.4) to determine all the logical disks on the node. For each mounted disk, it essentially executes the functionality of the *FPMount* primitive. It checks Page 0 for prefixes, enters these in the PMT, and creates an FDM arobject for each disk with a directory fragment. Once all entries are made to the PMT, all peer FPMMs are informed of the new entries.

There was a danger of deadlocks in the restart sequence, especially if multiple nodes happened to come up at the same time. To avoid deadlock, a separate Worker process is provided, which waits on the peers FPMMs. In addition, a timeout is associated with the *FPRequestTable* primitive to avoid indefinite blocking. If several nodes come up at the same time, each node can incorporate entries in its PMT pertaining to its own logical disks, and then send this information to the other nodes.

3.6.4 Directory Management

The main purpose of the Directory Management Subsystem is to maintain a part of the file directory, and thereby map filenames occurring in this directory fragment to their corresponding global page set identifiers. This mapping determines the logical disk on which the file resides, and the ID of the page set arobject which manages the root page set for the file. In addition to the mapping information, the directory also maintains static file information, such as creation date, modification date, and file length. An instance of the File Directory Management Subsystem (FDM Subsystem) exists for each disk with a directory fragment on it.

Operations which require directory mapping information, such as *FDOpen*, *FDClose*, *FDCreate*, and *FDDelete*, are all supported by the FDM subsystem. Filename matching operations, such as finding all files with a matching prefix, are also directory oriented operations, and are performed by this subsystem. In addition to these operations, primitives are provided for accessing and modifying the directory entries.

---

fa-aid = **FDOpen**(filename)
fa-aid = **FDCreate**(filename, type [, node])
action = **FDClose**(modified, filesize)
val = **FDDelete**(filename)
val = **FDUndeleteAtomic**(filename)
val = **FDExpungeAtomic**(filename)

val = **FDSync**([filename])
last = **FDFind**(prefix, after, comp, buffer, bufsize)

val = **FDInsert**(filename, file-info)
val = **FDGet**(filename, file-info)
val = **FDRemove**(filename)

val = **FDRestart**(diskid)
val = **FDUnmount**()

AID fa-aid — The ID of the File Arobject, corresponding to the file being opened or created.

FDACTION action — Specifies one of three possible actions to be taken: CONTINUE, DEACTIVATE, or KILL.

BOOLEAN val — TRUE if the specified operation is done successfully, otherwise FALSE.

FILENAME last — A number of components or tails (depending on the value of *comp*) of filenames are found, matching the given prefix. The last component or tail is returned in the *last* variable. If the end of the list of components or tails has been reached, a special value (NULL) is returned.

FILENAME filename
The name of the file for which the specified operation is to be performed.

FILETYPE type The type of file to be created: NORMAL, PERMANENT, or ATOMIC.

NODENAME node The ID of the preferred node on which the file should be created.

BOOLEAN modified
Flag which specifies whether the file has been modified or not.

INT filesize The size of the file in bytes.

FILENAME prefix The filename prefix which has to be matched, and corresponding to which all filename components or tails of filenames have to be returned.

FILENAME after The value returned by the variable *last* is used here. Hence it is either a component or tail of a filename. Matching of components or tails has to start after this element. If the value of *after* is NULL, matching starts from the first element.

BOOLEAN comp If the value is TRUE, only the next components of the filenames following the matching prefix are returned. If the value is FALSE, the entire tails of filenames are returned.

BUFFER *buffer The address for the data buffer.

INT bufsize The size of the buffer being provided.

FILEINFO file-info Returns all the information about a file saved in the directory entry.

DISKID diskid The Logical Disk ID of the disk volume being operated on.

The *FDOpen* primitive opens an existing file. If this is the first *FDOpen* operation on the file, the Arobject/Process Management Subsystem is called upon to activate the inactive file arobject, and return the ID of the active file arobject. If the file has already been opened, the count of opens is incremented, and the ID of the active file arobject is returned. If the file does not exist, an error is returned. The *FDCreate* primitive creates a file of the specified type (NORMAL, PERMANENT, or ATOMIC) on the preferred computing node, if possible. The Arobject/Process Management Subsystem is requested to create a file arobject of the appropriate type, and the ID of this arobject is returned. A directory entry for the newly created file is added. Following creation, the file is opened. If the file already exists, an error is returned.

The *FDClose* primitive is always invoked by the file arobject, when some client has requested the

FAClose operation. It responds by specifying one of three operations (CONTINUE, DEACTIVATE, or KILL), to be carried out by the file arobject. It also modifies the static file information saved in the directory. If the file has been modified, the modification date and time is changed to the present time, and the file length is updated.

The actions taken by the *FDDelete* primitive depend on whether the file is open or not at the time of the request. If the file arobject is inactive, *FDDelete* deletes the file and removes its directory entry. If the file is open at the time the *FDDelete* request is received, a clean file deletion is provided. Further *FDOpen* operations are not allowed, and once the file has been closed by all current users, it is deleted. In the case of atomic files, the *FDDelete* primitive merely flags the file (in the directory) as deleted, so that it can be recovered in case the transaction is aborted.

The *FDUndeleteAtomic* and *FDExpungeAtomic* primitives are provided to allow atomic deletion of files. The *FDDelete* primitive for an atomic file sets a flag in the directory entry. After the transaction commits, the garbage collector can invoke the *FDExpungeAtomic* primitive,[34] which removes the directory entry, and expunges the file. If, on the other hand, the transaction has to be aborted, the *FDDelete* can be undone by the *FDUndeleteAtomic* primitive, since the file has not been expunged.

The *FDSync* operation ensures that any buffered directory information pertaining to the specified file is synchronized with the version saved on disk. If a file name is not specified, the contents of the entire directory buffer are synchronized.

The purpose of the *FDFind* primitive is to allow some of the convenience of hierarchical directories. In a file system with a hierarchical directory, it is usually possible to search for all files and subdirectories which exist in a particular directory. The *FDFind* primitive implements the same notion in our "flat" file name space. Given a file prefix (parallel to the full pathname of a directory) it can find all the components (filenames or subdirectory names contained in the directory). It is also possible to find the full tails of all matching file names (parallel to doing a recursive directory search). The boolean variable *comp* determines which type of search is undertaken: for matching components or tails. Since the buffer for returning the matching elements may not be able to hold the entire list of matches found, the last matching element is returned as the result of the *FDFind* primitive. The search can begin after this element when the next *FDFind* operation is invoked.

Three primitives are provided for manipulating the directory entries. The *FDInsert* primitive allows a

---

[34] The *FDExpungeAtomic* operation can alternatively be invoked by the transaction manager, or even the client.

new entry to be inserted in the directory. The *FDGet* primitive allows the directory entry for a particular file to be read, and the *FDRemove* primitive removes the entry for a particular file. These primitives can be used by the Client Arobject Interface to provide several useful functions to the client. For example, the *RenameFile* primitive is implemented by first obtaining file information using *FDGet*, then removing the old directory entry with *FDRemove*, and finally adding a new directory entry with *FDInsert* to correspond to the new file name.

The *FDRestart* primitive is primarily an initialization operation, invoked when the FD arobject is created. A cleanup operation on the logical disk associated with this FD arobject instance is initiated. The directory is checked for all normal files, and the Page Set Subsystem is asked to free all page sets corresponding to these files. Their directory entries are also removed. The *FDUnmount* operation is invoked when the logical disk associated with this FDM instance has to be unmounted cleanly. Once an *FDUnmount* has been received, FDM does not accept any new operations except *FDClose*. When all currently open files have been closed, it sends a reply to the requestor of *FDUnmount*, and destroys itself.

3.6.4.1 Components of the Directory Management Subsystem

The FDM Subsystem consists of three main components: (1) the Directory Manager process, (2) the buffer for the directory B-tree, and (3) the Open Files Table. In addition to these components, a few other items of information are also maintained, such as the logical disk ID, and some flags. The three components are shown in Figure 3-18, and described briefly in this section.

The Directory Manager process is responsible for providing all of the functionality of the FDM subsystem. It accepts all requests for FDM operations, and returns the results. It manages all the FDM data structures as well. In order to maintain the directory buffer, it invokes operations on the Page Set Subsystem.

The file system directory uses a B-tree structure, and is saved on disk as a number of different page sets (refer Section 3.6.4.2 for details). Some of the pages of the directory are buffered in main memory by the FDM subsystem, in a data structure known as the Directory Buffer. The FDM Manager Process interacts with the Page Set Subsystem in obtaining directory pages from disk, and in writing back any dirty buffered pages.

The Open Files Table keeps track of the files which are open. It maintains the names of these files, their arobject IDs, and the number of outstanding opens on each file. In addition, it maintains some flags, such as for open files which are deleted (or expunged). Whenever an *FDOpen* request is

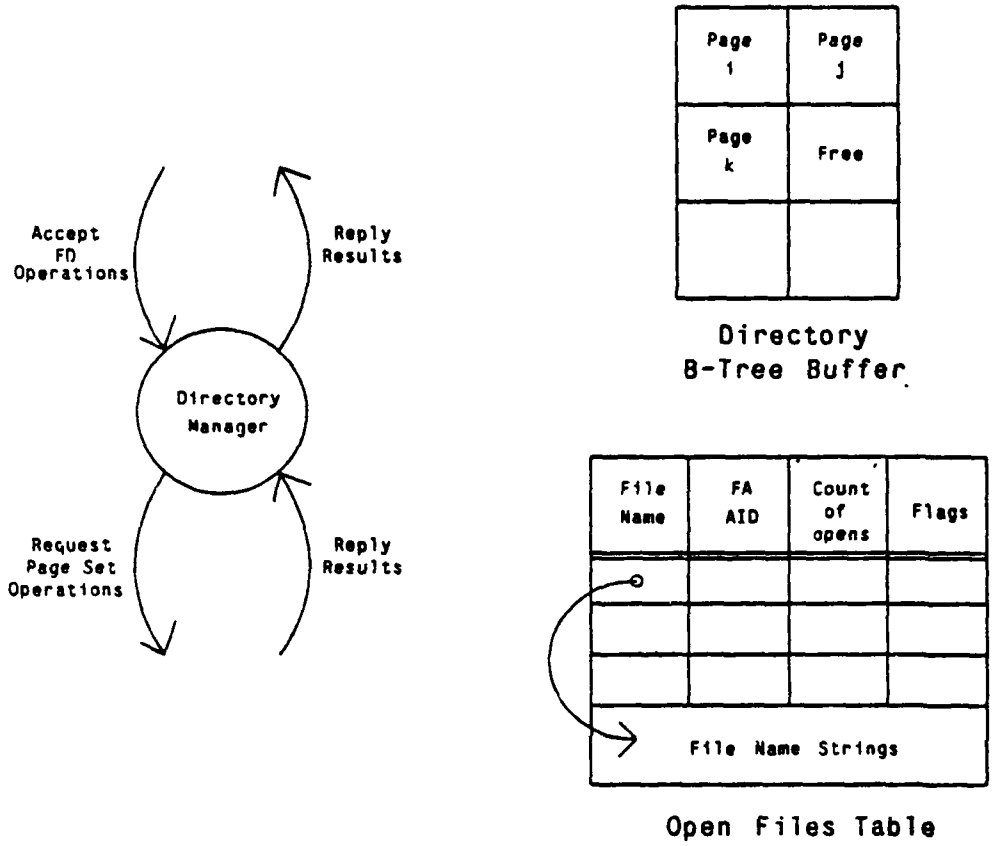


Figure 3-18: Components of the Directory Management Subsystem

received, the OFT is first checked to see whether the file has already been opened or not. When a file is opened for the first time, an entry is made in the OFT, but for subsequent opens, the count of outstanding opens is incremented. For each *FDClose* operation, the count of outstanding opens for that file is decremented. If the count becomes zero, the entry is removed from the OFT, and the file arobject is told to deactivate itself. If an *FDDelete* operation is received on an open file, a DELETE flag is set in the OFT. Further *FDOpen* operations are not allowed, but *FDClose* operations are accepted, to achieve a clean deletion. Once a DELETE flag has been set, the file arobject is told to kill itself at the time of the last close operation.[35]

[35] If an atomic file is deleted in this way, its file arobject is deactivated, and not killed. A flag is set in the directory entry for the file, indicating that it has been deleted, and further operations on that file, other than FDUndeleteAtomic or FDExpungeAtomic, are not allowed.

3.6.4.2 The Directory B-Tree

The file system directory maps file names to global page set IDs. It also saves some static information for each file: the date and time of creation and last modification, the file type, the file length, and some flags (e.g. the DELETED flag, to mark files which have been deleted but not expunged). Logically, the file system directory fragment on a particular disk is an independent B-tree. The entries in this B-tree are ordered by the filename (in alphabetical order). Each node of the directory B-tree is implemented as a page set. Thus, the directory is implemented as a set of page sets, which are maintained on disk by the Page Set Subsystem.

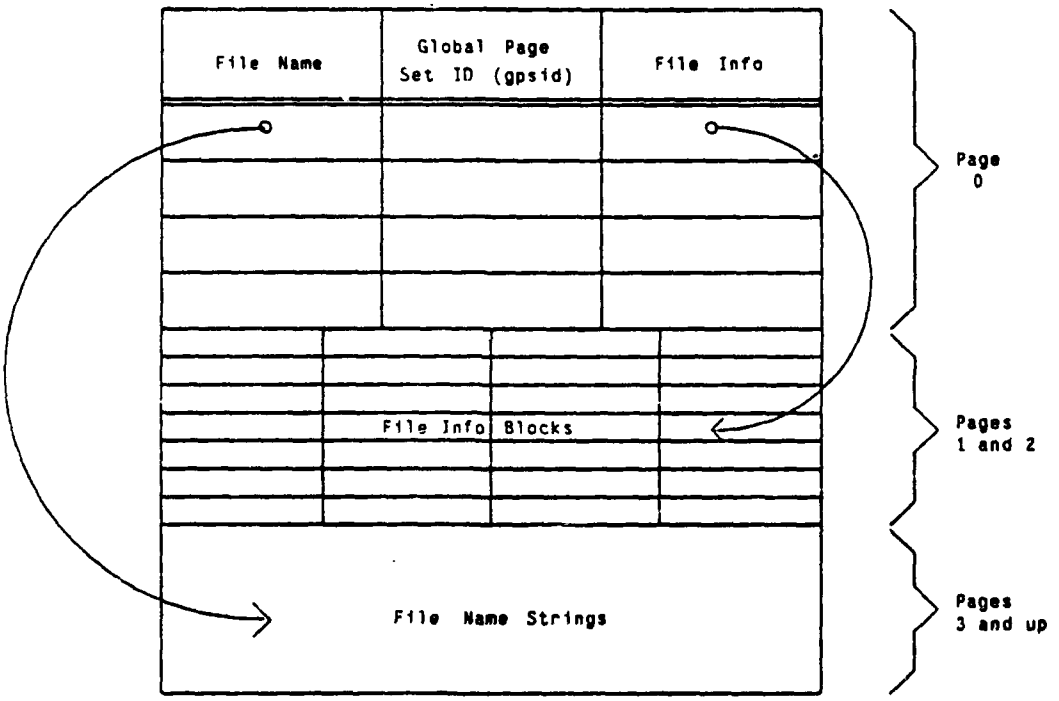


Figure 3-19: Structure of a Directory B-tree Node

The structure of a node in the directory B-tree is somewhat complex (refer Figure 3-19). Each node of the directory B-tree contains a large number of entries (200 to 250). Logically, each directory entry is a triple consisting of the filename, the global page set ID, and a pointer to the file information block. The actual implementation is complicated by the fact that file names have variable sizes. Hence, instead of keeping the file name as the first element of the triple, a pointer to the file name is kept. This file name pointer is a byte offset from the start of the area for saving the full file names.

Each node of the directory B-tree is a page set. The first page of this page set is the header page, which contains a series of triples. The first element of the triple is a pointer to the full file name, the second is the global page set identifier, and the third is a pointer to the information block (fixed size) for this file. If the node is internal (not a leaf node in the tree), the page set ID points to the node of the B-tree to be searched next. However, if the node is external (a leaf node), the page set ID refers to the location of the file (ID of the root page set for the file). The second and third pages of the node page set are reserved for the file information blocks. These pages will be unused in the case of internal nodes. The full file names are stored starting at the fourth page. Hence, the pointer to the file name holds a byte offset from the start of the fourth page.[36]

3.6.4.3 FDM File Manipulation Operations

In this section, we will briefly discuss the key interactions of the of FDM file manipulation operations: *FDOpen*, *FDClose*, *FDCreate*, *FDDelete*, *FDUndeleteAtomic*, and *FDExpungeAtomic*. Each of these operations (except *FDClose*) is invoked by some Client Arobject Interface, and the result is returned there.

When the operation *FDOpen* is requested, the Open Files Table is first checked to see whether the file has already been opened. If this is indeed the case, the arobject ID for the file can be determined from the OFT. No further action is required, except the updating of the count of opens for the file. However, if this is the first open request, the inactive file arobject has to be activated. First the directory B-tree is searched (by bringing appropriate directory pages into the Directory Buffer) to find the global page set ID (GPSID) for the (root page set of) file in question. Then the Arobject/Process Management Subsystem is called to activate the arobject for the given page set, and return the file arobject ID. An entry for this file arobject is now made in the OFT. Also, the newly created file arobject is initialized by invoking the *FARestart* primitive. This completes the *FDOpen* sequence, and the new file arobject ID is returned to the FDM subsystem.

The *FDCreate* sequence is similar in flavor to the *FDOpen* sequence. First the directory B-tree is checked to be sure that the file does not already exist. Next, the Arobject/Process Management Subsystem is invoked to create a new file arobject of the specified type. The A/PM also decides where the new file is to be placed[37]. The GPSID and the arobject ID for the newly created file arobject

---

[36] For efficiency reasons, the file name pointer may actually consist of two parts: a *prefix* pointer, and a last *component* pointer. Since file names are typically long, and many of the file names for a given directory node differ only in the values of their last components, this scheme of storing names should significantly reduce directory storage requirements.

[37] The file placement algorithm is subsumed by the arobject and process placement algorithms, and is implemented by the Arobject/Process Management Subsystem. If a preference for placement is specified, this is taken into account in making the final decision.

are returned by A/PM. An entry for the file is created in the directory, and in the OFT. Finally, the new file arobject is "restarted" (with *FARestart*), and its ID is returned to the FDM subsystem.

The *FDClose* primitive is always invoked by the file arobject, when a client has requested an *FAClose* operation. The FDM subsystem responds to this operation by specifying one of three actions for the file arobject: CONTINUE, DEACTIVATE, or KILL. When *FDClose* is requested, first the count of outstanding open requests in the OFT is decremented. If some outstanding opens still remain, the CONTINUE command is given to the file arobject. If however, there are no outstanding opens, the entry for this file in the OFT is deleted. If the file has been modified (*modified* is TRUE), the information blocks for the file are updated. Finally, the file arobject is asked to deactivate itself. The last *FDClose* operation on a file is somewhat different if the file has already been flagged for deletion by some other client. The directory entry for the file is removed, and the file arobject is asked to kill itself (which automatically removes the associated page sets).

The implementation of the *FDDelete* primitive is quite straightforward. If the file arobject is inactive when the primitive is invoked, the Arobject/Process Management Subsystem is called to destroy the inactive file arobject (which removes the relevant page sets), and the directory entry for the file is removed. If the file is active at the time *FDDelete* is called, the delete flag is set for the file in the OFT. At the time of the last *FDClose* operation, the file is deleted by destroying the active file arobject.

The handling of the primitives (described above) for atomic files may be somewhat different. Creating, deleting, opening, and closing files atomically is a part of the bigger and more general problem of creating, deleting, activating, and deactivating arobjects atomically. Hence, these issues have to be addressed in a unified manner. At present, some hooks have been provided, which help in treating these primitives (especially *FDDelete*) as compound transactions with compensation actions. Thus, when an atomic file is to be deleted, the directory is merely flagged as deleted, but the arobject is not destroyed. Once the highest level transaction has committed, the garbage collector can expunge all "deleted" files, by calling *FDExpungeAtomic*. If the transaction has to abort, the deleted file can be reclaimed by calling *FDUndeleteAtomic*.

3.6.5 File Management Scenarios

In this section, we will show how the four subsystems within the File Subsystem interact with one another, and with other subsystems in the kernel, to provide the specified functionality. We will examine the operations most frequently encountered in the lifetime of a file, and thereby explain the interactions within the File Subsystem.

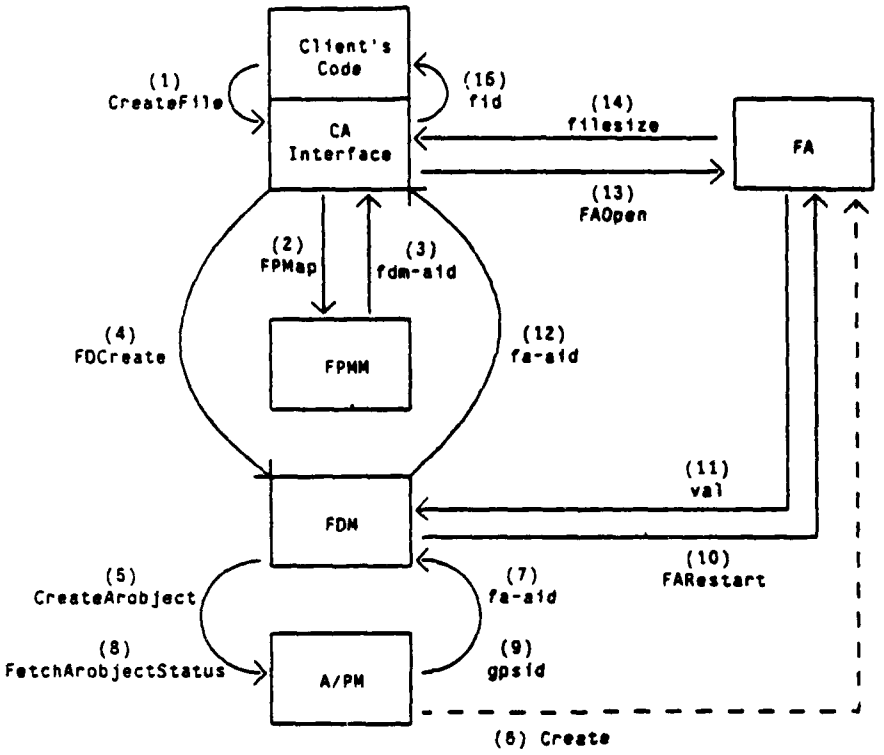


Figure 3-20: Interactions in the Implementation of the CreateFile Primitive

The *CreateFile* primitive is invoked by the client when a new file has to be created. In response to this request, the Client Arobject Interface has to first determine the ID of the Directory Management arobject which will manage the directory fragment corresponding to the new file. The Client Arobject Interface first invokes the *FPMap* primitive (refer Figure 3-20). The *FPMap* operation is accepted by the FPMM Manager process, which checks the prefix of the filename in the Prefix Map Table, to determine the ID of the corresponding FDM arobject (let this value be *fdm-aid*). The result of the *FPMap* operation (*fdm-aid*) is returned to the Client Arobject Interface. The CA Interface now invokes the *FDCreate* operation on the correct instance of the Directory Management Arobject (*fdm-aid*). The FDM Manager reads relevant pages of the directory into the buffer, and ensures that the file does not already exist. It then calls the Arobject/Process Management Subsystem to create a file arobject (using the *CreateArobject* primitive). The Arobject/Process Management Subsystem returns the ID of the newly created file arobject. Next, the *FetchArobjectStatus* primitive is used to determine the global page set ID for the root page set corresponding to this arobject. The Directory Manager then makes an entry in the directory for the new file, showing the mapping of its filename and its GPSID. It

writes the creation date in the file information block. It also invokes the *FARestart* operation on the newly created file arobject to initialize it, and provide some information such as the file size. The FDM Manager makes an entry for the file in the Open Files Table, and returns the ID of the file arobject to the Client Arobject Interface. The CA Interface calls the File Arobject with an *FAOpen* request. The open operation is carried out, and the current filesize is returned. The CA Interface then returns to the client with the ID of the file.[38]

The interactions of the *OpenFile* primitive are quite similar to those of the *CreateFile* primitive. First, the *FPMap* primitive is invoked by the Client Arobject Interface. Then, the *FDOpen* operation is invoked. If the file has not already been opened by some other client, the FDM subsystem reads the directory into the buffer to determine the global page set ID for the file. It then invokes the Arobject/Process Management Subsystem to activate the inactive file arobject corresponding to the GPSID. The active file arobject is initialized by *FARestart*, and an entry is made in the Open Files Table. The file arobject ID is returned to the CA Interface subsystem. Finally, *FAOpen* is called, and the interactions of *OpenFile* are complete.

The interactions in *ReadFile* and *WriteFile* are quite straightforward. The *ReadFile* and *WriteFile* operations given by the client are buffered by the CA Interface. The *FARead* and *FAWrite* operations are invoked by the CA Interface as necessary. This reduces the number of interactions with the file arobject.

The *CloseFile* operation given by the client is translated to *FAClose* by the CA Interface (refer Figure 3-21), after first flushing the CA Interface buffers. The file arobject removes the client from the Open Client List, and invokes the *FDClose* operation. The information block for the file is updated, and the file is closed as explained in Section 3.6.4. The file arobject is told to CONTINUE, DEACTIVATE, or KILL itself, depending on other outstanding open requests, and whether the file has been deleted by some other client. The file arobject replies to the *FAClose* operation, and then executes the specified action. If it has to deactivate or kill itself, it calls on the Arobject/Process Management Subsystem to perform the necessary operation.

The interactions for *DeleteFile* are quite simple. The *DeleteFile* operation is translated to *FDDelete* by the CA Interface, after first using the *FPMap* operation if necessary. The FDM subsystem calls the Arobject/Process Management Subsystem to destroy the inactive file arobject, and then removes its directory entry. If the file is open at the time the operation is invoked, a DELETE flag is set in the Open Files Table, and the deletion is completed at the time of the last *FDClose* operation.

[38] The *file ID* returned to the client arobject is different from the ID of the corresponding file arobject. The CA Interface maintains a mapping between the longer file arobject ID, and the shorter file ID which it generates.

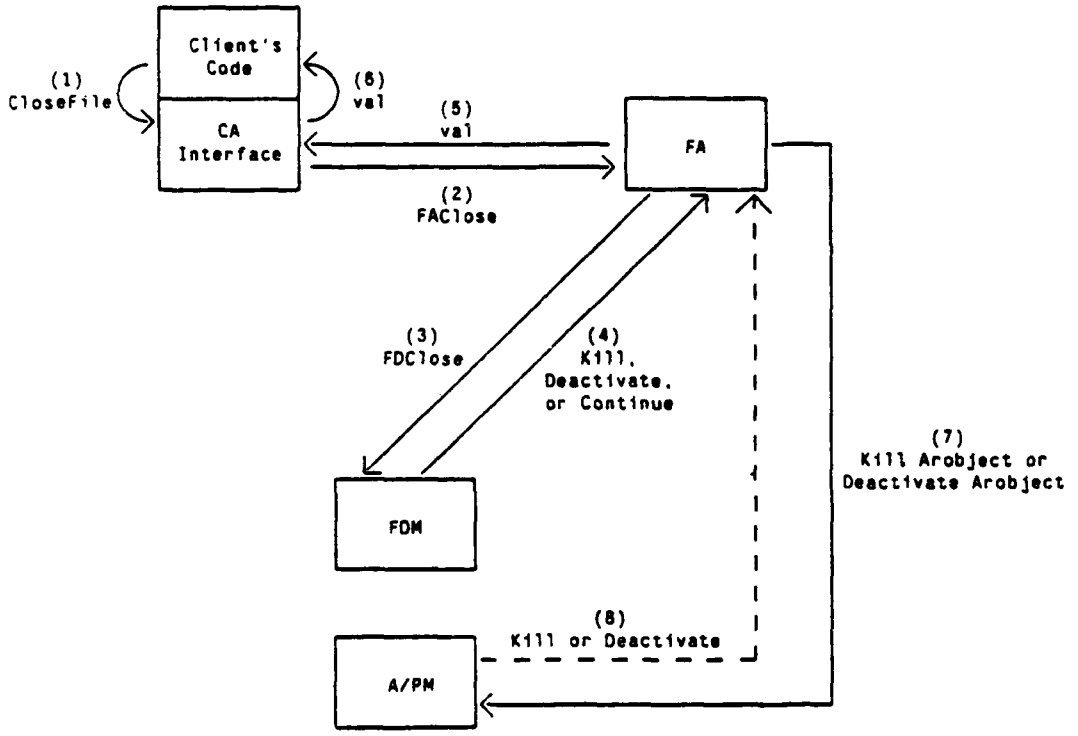


Figure 3-21: Interactions in the Implementation of the CloseFile Primitive

3.7 Page Set Subsystem

The main purpose of the Page Set Subsystem is to provide an abstract interface to the physical secondary memory storage devices (disks). The abstraction provided is that of a set of *logical disks*, each of which holds some number of *page sets*. A logical disk is a contiguous region (partition) of a physical disk, consisting of *N* logical pages, numbered from 0 to *N*-1.[39] Each physical disk can contain at most one ArchOS logical disk. However, the logical disk can be located anywhere on the physical disk, and can possibly cover the entire disk. Each logical disk has a unique identifier (logical disk ID) permanently associated with it, i.e. it will always have the same logical disk ID, regardless of which node or disk drive the disk is mounted on.

A page set is (conceptually) an infinite set of pages, numbered 0, 1, 2, ... Any pages of a page set

[39] Currently a page is defined to be 2K bytes in size, but that is a parameter which can be changed and tuned to obtain "optimum" performance.

which have not been explicitly written, are defined as containing all zeros.[40] Pages can be read or written in any order within a page set, i.e. the pages of a page set are randomly accessible, and sparse page sets are permitted. All of the pages comprising a single page set will be located on the same logical disk. This helps reduce the number of page sets which would be affected by any single disk or node failure. Every page set is given a page set ID, which is unique within the logical disk on which it resides. A globally unique page set identifier (GPSID) can then be constructed by combining the logical disk ID with the page set ID.

ArchOS supports two different classes of page sets: *standard page sets*, and *atomic page sets*. Standard page sets themselves come in three types: *temporary*, *permanent*, and *dual*. Temporary page sets are intended to provide short term storage for data on secondary memory. They are destroyed, and their pages automatically reclaimed by the system, following a crash. Temporary page sets are primarily used as the paging areas for the "volatile" segments of process address spaces, such as the User Stack, User Heap, and Shared Normal Segments (see Section 3.2.2.5). In particular, normal file arobjects (see Section 3.6) store all of their data in volatile segments, and hence in temporary page sets.

Permanent page sets are primarily used to store *permanent* abstract data type instances, such as permanent files. Permanent page sets survive system crashes, but their consistency following a crash is not guaranteed. If a permanent page set was being actively modified at the time of a crash, some of the modifications may be permanently recorded while others are lost. Furthermore, if a page was being written precisely at the time of the crash, that page could be written incorrectly, resulting in the loss of some data. Note that permanent page sets closely resemble the notion of "files", as commonly found in more conventional operating systems.

Dual page sets are like permanent page sets, except that each page is written atomically. If a crash occurs while writing a page in a dual page set, either the new contents of the page will be permanently and correctly recorded, or the original contents will be recovered (as if the write operation had never occurred). The atomic writing of an individual page is accomplished by carefully writing two copies of the page (hence the name "dual" page set). For more details, see the explanation of "stable storage" in [Sturgis 80]. Dual page sets are used for storing key data structures, such as file system directories, for which it is important to minimize the amount of damage in the event of a crash.

---

[40] Of course, such unwritten pages do not occupy any space on the logical disk.

The final type of page set supported by ArchOS is the atomic page set. Atomic page sets actually represent a second major class of page sets, differing (somewhat) from standard page sets, both in terms of their implementation and their user interface. Atomic page sets permit the grouping of operations into atomic transactions, (either all of the operations will be completed, or none of them will be). The operations comprising a single transaction can involve multiple atomic page sets, stored on any number of different logical disks, and spanning any number of system nodes. Furthermore, operations are not restricted to being page-oriented. Instead, arbitrary sequences of bytes can be read or written. Thus, two separate transactions can modify different parts of a single page, without interference or unexpected inconsistencies arising.

Atomic page sets also support the notion of nested transactions. Both elementary and compound transactions can be arbitrarily nested, as explained in Section 3.5. The *commit* or *abort* of a (sub)transaction will be properly reflected in the contents of the affected atomic page sets. To support this, the atomic page sets contain the intermediate states of all of the incomplete, nested transactions, in addition to the "current" page set contents. Atomic page sets provide complete failure atomicity for "soft and clean" failures, as defined in [Bernstein 83]. However, the consistency of atomic page sets in the face of "concurrent" updates from separate transactions is not automatically guaranteed. The lock management primitives of the Transaction Subsystem must be used to ensure such consistency (see Section 3.5). Atomic page sets are used to store *atomic* abstract data type instances, such as atomic files.

The Page Set Subsystem is implemented using four different types of kernel arobjects. Each node of the distributed computer system will contain one or more instances of these four arobjects, as illustrated in Figure 3-22. For each physical disk that is mounted on a node, a corresponding instance of the Logical Disk Subsystem, the Standard Page Set Subsystem, and the Atomic Page Set Subsystem will be created. In addition, each node will contain a single instance of the Page Set Restart/Reconfiguration Subsystem. Figure 3-22 shows the "uses" relationships among these component subsystems.

The Logical Disk Subsystem builds the logical disk abstraction (discussed above) on top of the available physical disk hardware. It manages the allocation of pages on the logical disk, and supports multi-page read/write operations to/from either local or remote primary memory (buffer) areas. The Standard Page Set Subsystem provides support for the three types of standard page sets: temporary, permanent, and dual. It constructs these page sets using the facilities of the corresponding Logical Disk Subsystem. Similarly, the Atomic Page Set Subsystem manages all of the atomic page sets that are stored on the corresponding logical disk. Finally, the Page Set Restart/Reconfiguration

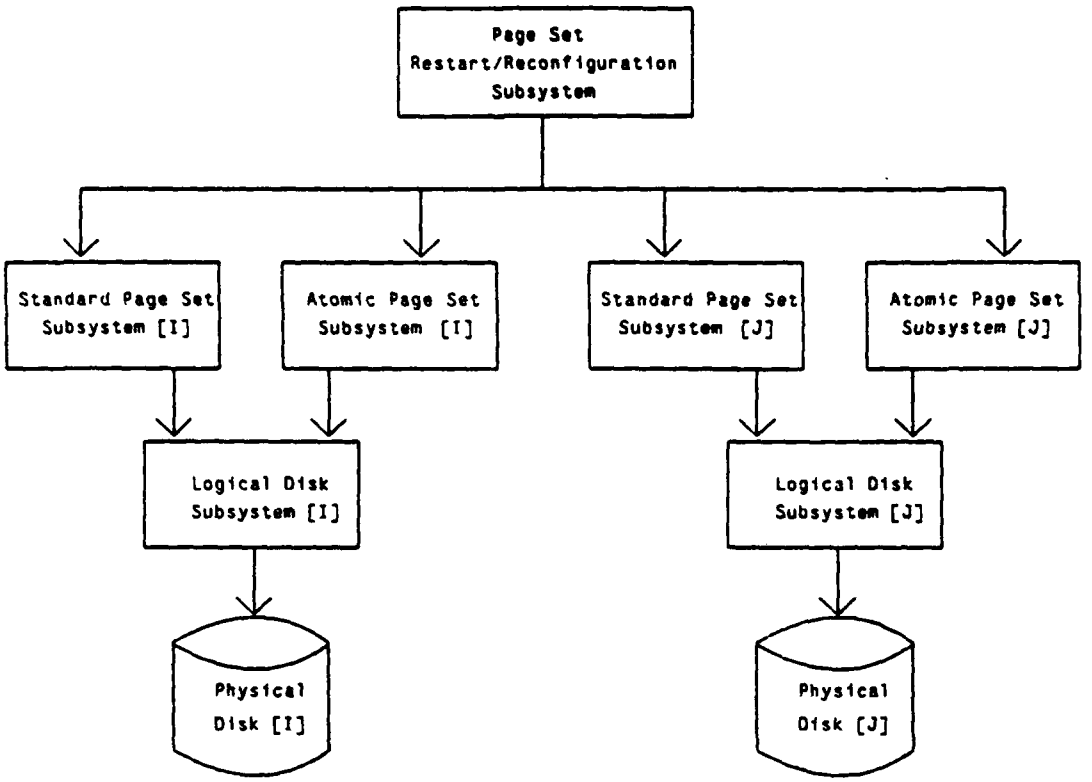


Figure 3-22: Major Components of the Page Set Subsystem

Subsystem (one per node) is responsible for constructing the three "per-disk" subsystems, for each disk that is mounted on its node. It allows disks to be "dynamically" added and removed from the system, without requiring the associated node to be completely restarted. In conjunction with this, the Page Set Restart/Reconfiguration Subsystem cooperates with its peers on other nodes, to maintain a global list (replicated at each node) of all the disks mounted anywhere within the distributed computer system.

3.7.1 Standard Page Set Subsystem

An instance of the Standard Page Set Subsystem kernel arobject is associated with each logical disk in the system. The Standard Page Set Subsystem supports three types of page sets: temporary, permanent, and dual. The same set of operations are provided for each of these types of page sets, but the semantics of some of the operations vary slightly, depending on the page set type. For example, the writing of pages in a dual page set is handled differently than in the case of a temporary or permanent page set. The Standard Page Set Subsystem supports multi-page read/write access to

the contents of page sets, allowing greater efficiency than would be possible with a page-at-a-time interface. The following is the complete list of primitives provided by the Standard Page Set Subsystem:

```
psid = PSCreate(type)
val = PSDestroy(psid)

npr = PSRead(psid, pnum, npages, buffer)
val = PSIsZero(psid, pnum, npages)
npw = PSWrite(psid, pnum, npages, buffer)
npw = PSZero(psid, pnum, npages)
val = PSMove(psid, pnum, npages, to-pnum)

val = PSSync([psid])
val = PSStatus(psid, statusbuffer)
val = PSRestart(disk-aid [, fast])
```

PSID psid — The identifier for the page set (unique within the logical disk). It includes an indication of the page set type: TEMPORARY, PERMANENT, or DUAL.

BOOLEAN val — TRUE if the specified operation is completed successfully; otherwise FALSE.

INT npr, npw — The actual number of pages read or written.

PSTYPE type — The type of page set to be created: TEMPORARY, PERMANENT, or DUAL.

INT pnum — The starting page number, within a page set, for the specified operation.

INT npages — The number of pages involved in the specified operation.

BUFFER *buffer — The buffer address in the kernel address space of either the local or remote node.

INT to-pnum — The destination page number, to which the specified pages are to be moved.

PSSTATUSBUFFER *statusbuffer — The buffer address for returning status information (can be either a local or remote kernel address).

AID disk-aid — The arobject ID of the Logical Disk Subsystem, associated with the disk to be used by this instance of the Standard Page Set Subsystem.

BOOLEAN fast — TRUE if this is to be a fast restart, avoiding all of the disk checking and garbage collection; otherwise FALSE. The default is FALSE.

The *PSCreate* primitive is used to create a new page set, of the specified *type*, on the disk associated with this instance of the Standard Page Set Subsystem. The identifier for the new page set (*psid*), which is unique within this logical disk, is returned. Initially, no pages are actually allocated to the page set, i.e. reading any page will return all zeros. The *PSDestroy* primitive frees all pages belonging to the specified page set (*psid*), and removes all record of that page set from the disk.

PSRead reads multiple (*npages*) pages, beginning with *pnum*, from page set *psid*, into the given primary memory *buffer*. If the buffer is located on a remote node, the required copying of data across the network will be handled automatically by the Logical Disk Subsystem (see Section 3.7.3). Any pages which have never been explicitly written will be read as all zeros. *PSRead* returns the number of (non-zero) pages which were actually read. *PSIsZero* can be used to check whether the specified range of pages, in the given page set, are all zero (unallocated). If so, it returns TRUE; otherwise FALSE.

PSWrite writes multiple (*npages*) pages from the given primary memory *buffer*, into page set *psid*, beginning at page number *pnum*. The number of pages actually written is returned. As with *PSRead*, the *buffer* can be located on either the local or a remote node. Any necessary cross-network data copying will be handled automatically by the Logical Disk Subsystem. *PSZero* provides a means for efficiently "zeroing" (deallocating) a range of pages within a page set. It is especially useful for "truncating the tails" of page sets, which is accomplished by zeroing the highest numbered (allocated) pages. The number of pages actually zeroed (i.e. the number of previously allocated pages which have now been deallocated) is returned.

PSMove can be used to "move" the specified range of pages within page set *psid*, to the new location specified by *to-pnum*. Overlapping source and destination ranges are permitted, with the result being equivalent to first deallocating all of the pages in the source and destination regions, followed by rewriting the original source pages into their destination locations. Among other things, *PSMove* can be used for "truncating the heads" of page sets, by specifying page zero as the destination, and moving all of the tail pages forward.

The *PSSync* primitive ensures that any buffered information within the Standard Page Set Subsystem is consistent (synchronized) with the information on secondary memory. If a particular page set (*psid*) is specified, only buffered information related to that page set is guaranteed to be consistent with the secondary memory information.[41] The *PSStatus* primitive returns (in *statusbuffer*)

---

[41] This may require much less time than synchronizing *all* of the buffered information.

information about the specified page set (*psid*). This includes the page set type, its maximum allocated page number (virtual size), and its actual number of allocated pages (physical size).

PSRestart initializes the Standard Page Set Subsystem, for the logical disk specified by *disk-aid*. It checks to ensure that the main secondary memory data structure (the Standard Page Set B-Tree, discussed below) is accessible and consistent, and ensures that the pages of any *dual* page sets are also consistent, by using the *DKRecoverDual* primitive of the Logical Disk Subsystem (see Section 3.7.3). A side effect of using *DKRecoverDual* is that all of the pages in the Standard Page Set B-Tree, and all of the dual page set data pages, will be marked as allocated. *PSRestart* then marks (as allocated) all of the data pages belonging to any *permanent* page sets stored on the disk (see the discussion of *DKMark* in Section 3.7.3). Finally, *PSRestart* "destroys" any *temporary* page sets which still reside on the disk, by carefully removing their entries from the Standard Page Set B-Tree.[42] *PSRestart* should only be invoked when the Standard Page Set Subsystem is first created by the Page Set Restart/Reconfiguration Subsystem (see Section 3.7.4 below), i.e. when the associated logical disk is first added to the system. The optional parameter (*fast*) can be used to indicate that a fast restart is in progress, and hence all of the consistency checks, page allocation marking, and removal of temporary page sets can be skipped. For more details concerning restart and garbage collection activities, see Section 3.7.4.

3.7.1.1 Components of the Standard Page Set Subsystem

Each instance of the Standard Page Set Subsystem kernel arobject (one per logical disk) has the simple structure illustrated in Figure 3-23. It consists of two main components: a single Manager process, and a B-Tree Buffer. The Manager handles all of the Standard Page Set primitives, discussed above. It uses the facilities of the associated Logical Disk Subsystem, in order to store, access, and manipulate the page sets recorded on its corresponding logical disk. Note that the Standard Page Set Subsystem itself does no buffering of data pages. It is left to the Logical Disk Subsystem to transfer data directly into and out of client buffers.

The B-Tree Buffer holds the most recently accessed nodes (pages) of the Standard Page Set B-Tree for this logical disk. The Standard Page Set B-Tree is the data structure used to record the sets of logical disk pages, comprising the standard page sets stored on this disk. The use of a B-tree [Comer

---

[42] It is assumed (unless a "fast" restart is in progress) that all of the pages on the logical disk have been "freed" prior to invoking *PSRestart*. As a result, the page allocation information must be reconstructed, and until it is, all requests to allocate new logical disk pages must be avoided. *PSRestart* must mark (as allocated) all of the pages which are part of the Standard Page Set Subsystem. Since none of the data pages from *temporary* page sets are marked, they are automatically freed. However, the record of the pages allocated to temporary page sets is still stored in the Standard Page Set B-Tree, and must be removed. The removal of entries from the B-Tree is a "safe" operation, since it does not require the allocation of any new logical disk pages, although it may involve the freeing of some pages from the B-Tree.

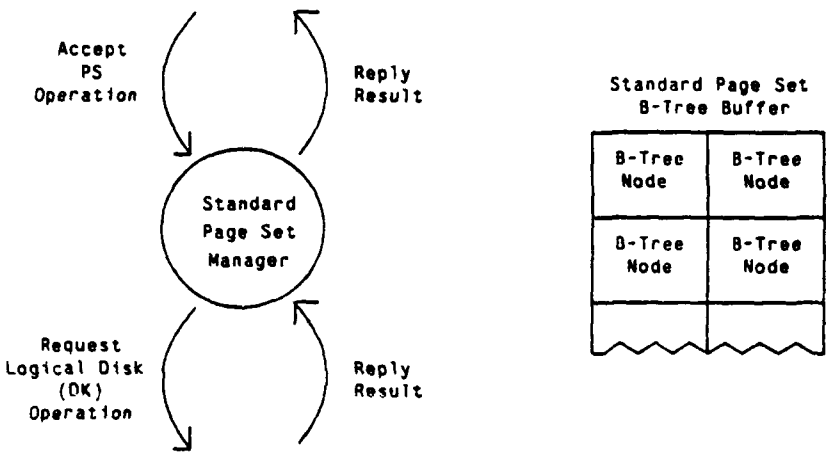


Figure 3-23: Components of the Standard Page Set Subsystem

79] for this purpose is quite similar to its use in the Xerox Distributed File System (XDFS) [Sturgis 80, Mitchell 82]. The logical structure of the Standard Page Set B-Tree is illustrated in Figure 3-24.

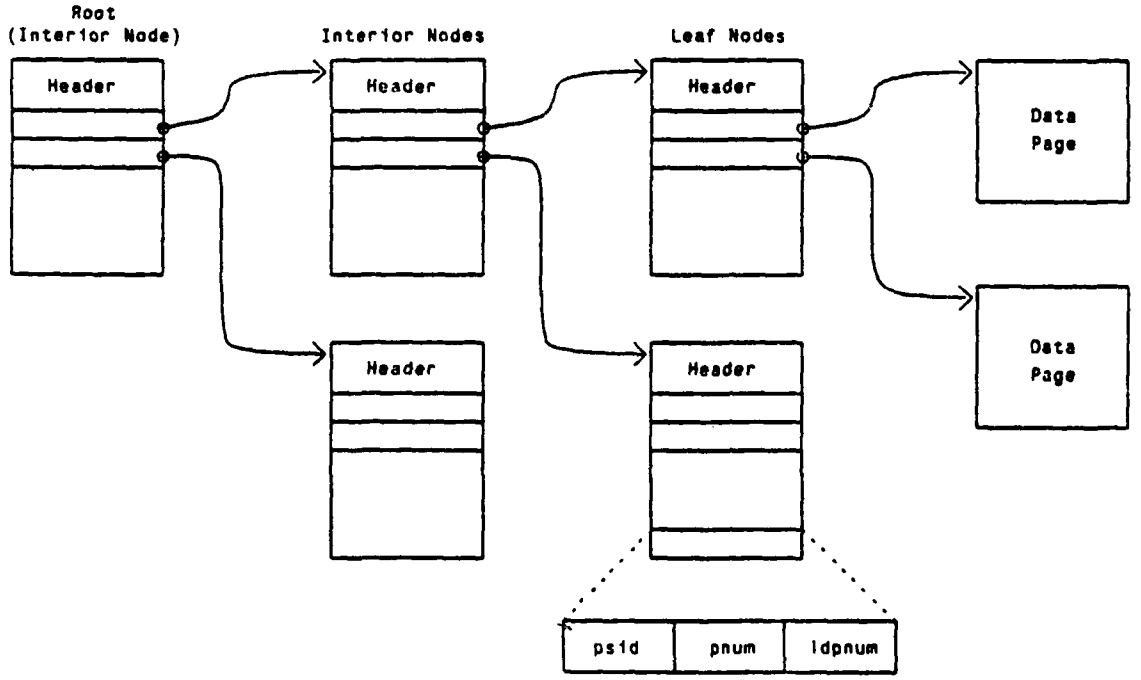


Figure 3-24: Standard Page Set B-Tree

In essence, the B-tree maintains a sorted list of data pages, ordered by page set ID (*psid*), and page number within *psid*. This allows the mapping from (*psid*, *pnum*) to the logical disk page number (ldpnum) to be be performed very quickly. Each node of the B-tree is stored as a *dual* logical disk page, in order to improve the reliability of the data structure. In addition to page mapping information, each node contains a small amount of header information, which indicates the type of node (interior or leaf), and the number of page map entries. Every map entry, whether in an interior or a leaf node, has the same structure (*psid*, *pnum*, *ldpnum*). However, the type of page pointed to by *ldpnum* will differ, depending on whether the node is interior or a leaf. A pointer to the root node of the B-tree (its *ldpnum*) is stored in page zero of the logical disk, so that the Standard Page Set Subsystem can always find its B-tree data structure.[43]

Note that the type of each page set (temporary, permanent, or dual) is included in the page set ID (*psid*). When a page set is first created, an entry for page number zero is added to the B-tree, with *ldpnum* set to zero to indicate that the page has not really been allocated yet. As pages are written or zeroed, appropriate entries are added or removed from the B-tree. However, there will always be an entry (whether allocated or not) for page zero of each page set that exists. Note that the structure of the B-tree allows all of the status information for each page set (its type, logical size, and physical size) to be determined quite easily. It also permits the pages of a page set to be written, read, and zeroed very efficiently.

In terms of storage overhead, the B-tree structure is also quite reasonable. If we assume that *psid*, *pnum*, and *ldpnum* are each 32 bits (4 bytes) long, then a single B-tree node (2K byte page) can contain 170 map entries, with 8 bytes remaining for header information. Since each leaf page of the B-tree is *dual*, and can map 170 data pages, only a little over one percent of the logical disk pages belonging to the Standard Page Set Subsystem should be needed for storing the B-tree itself. Furthermore, it should be noted that with 170 entries per node, a three level B-tree can map 170^3 data pages, which is considerably more than can be contained on any disk that is likely to be used as an ArchOS logical disk. Thus, the Standard Page Set B-Tree will be a maximum of three levels deep.

A special note should be made concerning the Standard Page Set B-Tree Buffer, illustrated earlier in Figure 3-23. Although (for clarity of exposition) each instance of the Standard Page Set Subsystem is shown as having its own buffer area, in practice, all instances on a single node would share a common Kernel Page Buffer Pool. Indeed, the Kernel Page Buffer Pool would be shared with many

---

[43] Page zero of each logical disk is a special *dual* page, which contains pointers to all of the major data structures on the disk. It also contains information about the disk itself, such as its size and its logical disk ID.

other subsystems on that node as well: Logical Disk Subsystems, Atomic Page Set Subsystems, File System Disk Directory Management Subsystems, and so on. Each subsystem allocates pages from the Kernel Page Buffer Pool (in least recently used order) as required. Pages are then returned to the pool (freed) as soon as they are no longer being actively used. Each subsystem maintains a list of the "free" pages in the buffer pool, which it has used recently. This list provides "hints" about pages in the buffer pool which may still contain information of use to the subsystem, i.e. they are buffer pages which can be reclaimed, rather than reading the information again from disk, assuming the pages have not already been reused by some other subsystem.

3.7.2 Atomic Page Set Subsystem

The Atomic Page Set Subsystem is quite similar to the Standard Page Set Subsystem, except that it provides the "atomic page set" abstraction. The primary difference between the standard and atomic page set abstractions is that the consistency of atomic page sets will be maintained in the event of a crash. In addition, the Atomic Page Set Subsystem allows arbitrary sequences of bytes to be read or written, rather than restricting the operations to being page-oriented. Atomic page sets are primarily used for storing *atomic* abstract data type instances, such as atomic files. As with the Standard Page Set Subsystem, an instance of the Atomic Page Set Subsystem is created for each logical disk in the distributed computer system.

The Atomic Page Set Subsystem provides primitives for "committing" arbitrarily nested elementary and compound transactions, in two phases: the prepare phase, and the commit phase. These two phases can be used by the Transaction Management Subsystem, as part of its three phase commit protocol (see Section 3.5). Using the mechanisms provided here, it is possible to atomically commit (or to abort) transactions which span multiple logical disks, and multiple nodes. It should be noted that the Atomic Page Set Subsystem only provides support for the properties of *failure atomicity*, and *durability* [Eswaran 76, Gray 81]. The sequencing and scheduling of concurrent transactions, so as to ensure *consistency*, is assumed to be handled by the Transaction Management Subsystem. The actual primitives provided by the Atomic Page Set Subsystem are the following:

```
psid = APSCreate(tid)
val = APSDestroy(tid, psid)

nbr = APSRead(tid, psid, location, nbytes, buffer)
val = APSIsZero(tid, psid, location, nbytes)
nbw = APSWrite(tid, psid, location, nbytes, buffer)
nbw = APSZero(tid, psid, location, nbytes)
val = APSMove(tid, psid, location, nbytes, to-location)

val = APSPrepareCommit(tid)
val = APSCommit(tid)
val = APSAbort(tid)

val = APSStatus(psid, statusbuffer)
val = APSRestart(disk-aid [, fast])
```

PSID psid — The identifier for the atomic page set (unique within the logical disk). It includes an indication of the page set type (ATOMIC).

BOOLEAN val — TRUE if the specified operation is completed successfully; otherwise FALSE.

INT nbr, nbw — The actual number of bytes read or written.

TID tid — The ID of the transaction to which this operation belongs. From the *tid* it is possible to determine the transaction type, as well as the parent transaction ID (if this is a nested transaction).

INT location — The starting location (byte offset from the beginning of the atomic page set) for the specified operation.

INT nbytes — The number of bytes involved in the specified operation.

BUFFER *buffer — The buffer address in the kernel address space of either the local or remote node.

INT to-location — The destination location, to which the specified bytes are to be moved.

APSSTATUSBUFFER *statusbuffer — The buffer address for returning status information (can be either a local or remote kernel address).

AID disk-aid — The arobject ID of the Logical Disk Subsystem, associated with the disk to be used by this instance of the Atomic Page Set Subsystem.

BOOLEAN fast — TRUE if this is to be a fast restart, avoiding all of the disk checking and garbage collection; otherwise FALSE. The default is FALSE.

The *APSCreate* primitive is used to create a new atomic page set, on the disk associated with this instance of the Atomic Page Set Subsystem. A transaction ID (*tid*) is specified, so that the new atomic page set will only come into permanent existence when transaction *tid* commits.[44] Until then, the new page set is only visible within this transaction, or within any nested subtransactions of *tid*. *APSCreate* returns the identifier for the new atomic page set (*psid*), which is unique within this logical disk. Initially, no pages are actually allocated to the page set, i.e. reading any part of it will return all zeros. The *APSDestroy* primitive frees all pages belonging to the specified atomic page set (*psid*), and removes all record of that page set from the disk. As was the case with *APSCreate*, the specified transaction ID (*tid*) determines when the effects of *APSDestroy* will be permanently committed, as well as the scope of their visibility in the interim.

APSRead reads *nbytes* bytes into the given *buffer*, beginning from byte *location* in atomic page set *psid*. The transaction ID (*tid*) is taken into account when determining which modifications (if any) to the specified bytes, by transactions which have not yet committed, should be visible to this read operation. If the buffer is located on a remote node, the data will be automatically copied across the network, using the special kernel *Copy* primitive. Any bytes which have never been explicitly written will be read as zeros. *APSRead* returns the number of (non-zero) bytes which were actually read. *APSIsZero* can be used to check whether the specified range of bytes, in the given atomic page set, are all zero. If so, it returns TRUE; otherwise FALSE. The specified transaction ID serves the same purpose as in the case of *APSRead*.

APSWrite writes *nbytes* bytes from the given *buffer*, into atomic page set *psid*, beginning at byte *location*. The number of bytes actually written is returned. The specified transaction ID (*tid*) determines when the effects of *APSWrite* will be permanently committed, as well as the scope of their visibility in the interim. As with *APSRead*, the *buffer* can be located on either the local or a remote node. Any necessary cross-network data copying will be handled automatically, using the special kernel *Copy* primitive. *APSZero* provides a means for efficiently "zeroing" a range of bytes within an atomic page set. The specified transaction ID serves the same purpose as in the case of *APSWrite*. When complete pages are zeroed, they are removed from the page set (deallocated). *APSZero* is especially useful for "truncating the tails" of atomic page sets, which is accomplished by zeroing the highest numbered (non-zero) bytes. The number of bytes actually zeroed (i.e. the number of previously non-zero bytes which have now been zeroed) is returned.

---

[44] If *tid* is a nested elementary transaction, the new atomic page set will only come into permanent existence when the top-level parent of *tid* commits.

APSMove can be used to "move" the specified range of bytes within atomic page set *psid*, to the new location specified by *to-location*. The transaction ID (*tid*) serves the same purpose as in the case of *APSWrite* and *APSZero*. Overlapping source and destination ranges are permitted, with the result being equivalent to first zeroing all of the bytes in the source and destination regions, followed by rewriting the original source bytes into their destination locations. Among other things, *APSMove* can be used for "truncating the heads" of atomic page sets, by specifying location zero as the destination, and moving all of the tail bytes forward.

The *APSPrepareCommit* primitive is used in phase one of the two phase transaction commit sequence for atomic page sets. All modifications to atomic page sets on this logical disk, which were made in the course of the specified transaction (*tid*), are carefully recorded in a special "Commit List" on secondary memory. These modifications can no longer be lost in the event of a crash. Transaction *tid* is flagged (on the disk) as "prepared to commit". The invoker is then informed (through the return value, *val*), of the Atomic Page Set Subsystem's readiness to commit the transaction. Following successful completion of the *APSPrepareCommit* primitive, the only allowed operation involving transaction *tid* is *APSCommit* or *APSAbort*. Note that *APSPrepareCommit* need not be invoked in the case of a nested elementary subtransaction (it has no effect). Such a subtransaction does not actually modify the permanent contents of the atomic page sets, until its top level parent transaction commits. Hence, a two phase commit sequence is unnecessary, since any failure will cause the parent transaction, as well as this "committed" subtransaction, to be aborted.

APSCommit is used in phase two of the two phase transaction commit sequence. It is also the only primitive required to "commit" a nested elementary subtransaction. In this latter case, *APSCommit* does not actually modify the permanent contents of the atomic page sets. Instead, it simply makes all of the modifications which were made in the course of the specified transaction (*tid*) visible to the parent of *tid*. When *APSCommit* is applied to a top level elementary transaction, or to any compound transaction, the specified transaction (*tid*) must already have been "prepared" (with *APSPrepareCommit*), in phase one of the commit sequence. In this case *APSCommit* is the second (final) phase of the commit sequence, and it is responsible for carefully updating the "permanent" contents of the atomic page sets, based on the modification information contained in the Commit List. *APSCommit* first flags (on the disk) transaction *tid* as "committed". It then returns the result (*val*) to the invoker, allowing the invoker to continue its execution while the Atomic Page Set Subsystem makes the required modifications to the atomic page sets.

The *APSAbort* primitive is used to abort the specified transaction (*tid*). This involves removing (undoing) all modifications made within transaction *tid*, or any of its nested subtransactions (except

already committed compound subtransactions). Note that transactions which have been "prepared", but not yet committed, can still be aborted. In that case the aborted modifications must be carefully removed from the Commit List.

The *APSStatus* primitive returns (in *statusbuffer*) information about the specified atomic page set (*psid*). This includes its virtual size (maximum non-zero byte location), physical size (actual number of allocated bytes), and an indication of which uncommitted transactions, if any, are still operating on this page set. Note that since no transaction ID is specified with the *APSStatus* primitive, the sizes reported are those of the current, "permanent" version of the atomic page set.

APSRestart initializes the Atomic Page Set Subsystem, for the logical disk specified by *disk-aid*. It checks to ensure that the main secondary memory data structures (the Atomic Page Set Permanent and Commit B-Trees, discussed below) are accessible and consistent, by using the *DKRecoverDual* primitive of the Logical Disk Subsystem (see Section 3.7.3). A side effect of using *DKRecoverDual* is that all of the pages in the Permanent and Commit B-Trees will be marked as allocated. *APSRestart* then marks (as allocated) all of the data pages belonging to any *atomic* page sets stored on the disk, and also marks all of the data pages associated with the Commit List (see the discussion of *DKMark* in Section 3.7.3).[45] Finally, *APSRestart* completes the processing of any transaction which is in the Commit List, and flagged as "committed". This involves carefully updating the permanent contents of the atomic page sets, based on the modification information contained in the Commit List.[46] *APSRestart* should only be invoked when the Atomic Page Set Subsystem is first created by the Page Set Restart/Reconfiguration Subsystem (see Section 3.7.4 below), i.e. when the associated logical disk is first added to the system. The optional parameter (*fast*) can be used to indicate that a fast restart is in progress, and hence all of the consistency checks and page allocation marking can be skipped. However, any remaining commit processing must still be performed, even in the case of a fast restart. For more details concerning restart and garbage collection activities, see Section 3.7.4.

3.7.2.1 Components of the Atomic Page Set Subsystem

Each instance of the Atomic Page Set Subsystem kernel arobject (one per logical disk) has the general structure illustrated in Figure 3-25. It consists of five main components: a single Manager process, three B-Tree Buffers, and a Data Page Buffer. The Manager handles all of the Atomic Page

---

[45] At this point, since *PSRestart* is assumed to have been invoked prior to *APSRestart*, all of the page allocation information for the logical disk has been reconstructed. Hence, it is now safe to allocate new logical disk pages, as required, without danger of reallocating pages that are already in use.

[46] Note that commit processing can be done "in the background" (after returning the result *val* to the invoker), just as in the case of *APSCommit*.

Set primitives, discussed above. It uses the facilities of the associated Logical Disk Subsystem, in order to store, access, and manipulate the page sets recorded on its corresponding logical disk.

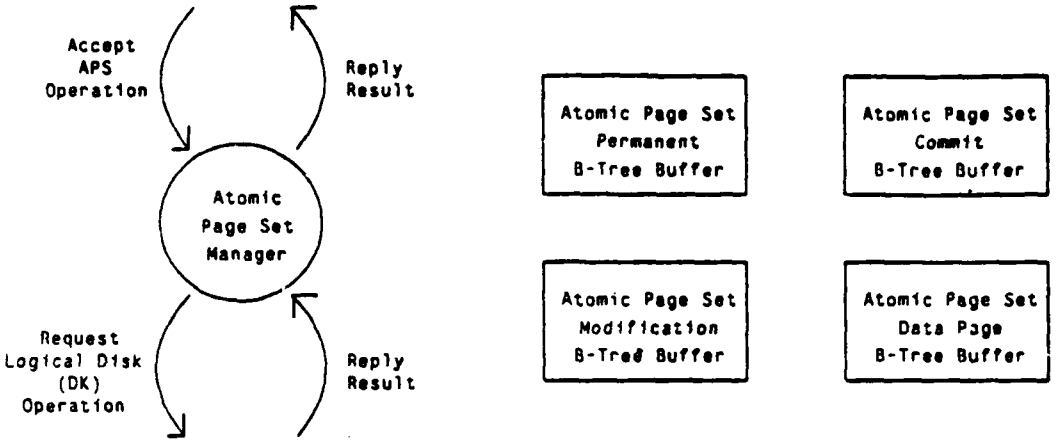


Figure 3-25: Components of the Atomic Page Set Subsystem

The four buffers in Figure 3-25 are primarily shown for clarity of exposition. In practice (as explained in Section 3.7.1), each instance of the Atomic Page Set Subsystem would *not* have its own, separate buffer areas. Instead, it would allocate buffer pages from a common Kernel Page Buffer Pool, and maintain "hints" lists concerning those pages which it most recently returned to the pool. For our purposes, however, it is more convenient (and not entirely inaccurate) to regard each instance of the Atomic Page Set Subsystem as having its own buffer areas.

The Permanent B-Tree Buffer holds the most recently accessed nodes (pages) of the Atomic Page Set Permanent B-Tree, for this logical disk. The Permanent B-Tree is the data structure used to record the sets of logical disk pages, comprising the current, permanent versions of the atomic page sets, stored on this disk. The logical structure of the Permanent B-Tree is identical to that of the Standard Page Set B-Tree, as discussed at length in Section 3.7.1, and illustrated in Figure 3-24. Refer to that Section for details. However, updating the Permanent B-Tree must be done more carefully than in the case of the Standard Page Set B-Tree, since in this case the update must be done atomically with respect to system failures. The technique for carefully (atomically) updating the Permanent B-Tree is outlined briefly in the discussion of transaction commit handling, below. As with the Standard Page Set B-Tree, a pointer to the root node of the Permanent B-tree (its *ldpnum*) is stored in page zero of the logical disk, so that the Atomic Page Set Subsystem can always find it.

The Modification B-Tree Buffer holds the most recently accessed pages of the Atomic Page Set Modification B-Tree. The Modification B-Tree is a temporary data structure, which contains the list of modifications that have been made to atomic page sets, in the course of the currently active, uncommitted transactions. Although the modification list is usually expected to be quite short, storing it as a B-tree will allow it to grow arbitrarily large, when needed. The general structure of the Modification B-Tree is similar to that of the Permanent B-Tree and the Standard Page Set B-Tree (see Figure 3-24). However, the nodes of the tree are stored in normal (rather than dual) pages, since this is a temporary data structure that will be "thrown away" automatically following a crash. The structure of a leaf node of the Modification B-Tree, along with its associated data pages, is illustrated in Figure 3-26.

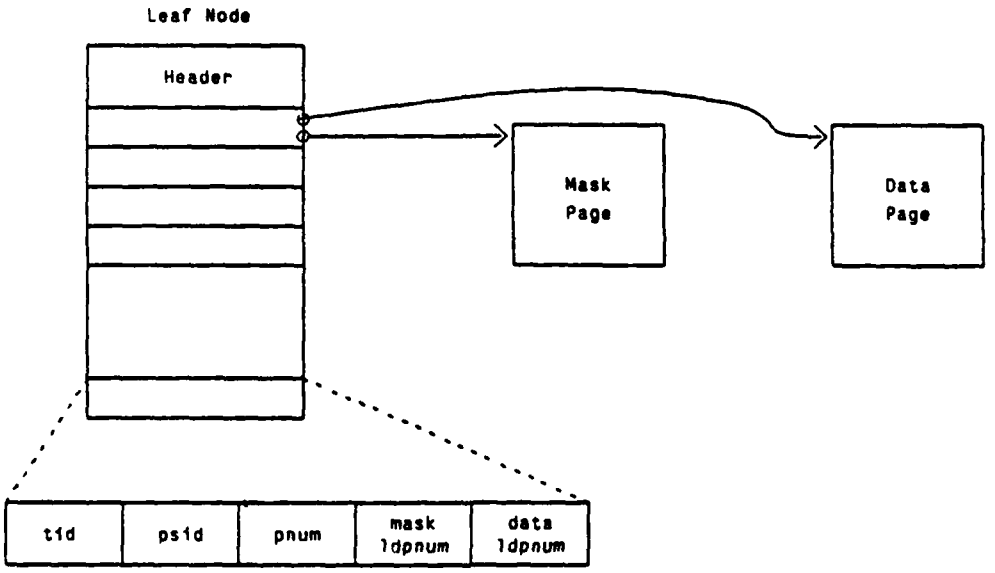


Figure 3-26: Atomic Page Set Modification List

The Modification List (B-Tree) is ordered by (*tid*, *psid*, *pnum*), and allows the mapping from these triples to the associated page modifications, to be performed very quickly. Page modifications are represented by a Data Page, containing the new values for the modified bytes within the page, and a Mask Page, which indicates which bytes have been modified. Modified bytes are indicated by a corresponding Mask Page byte with hexadecimal value FF. Unmodified bytes have value zero in both

the Data Page and the Mask Page.[47] If the entire Data Page has been modified, then the Mask Page can be omitted (*mask ldpnum = 0*). If the entire Data Page has been zeroed, then the Data Page can also be omitted (*data ldpnum = 0*). One other special case is that of atomic page sets that have been destroyed. In this case the Modification List contains only an entry for page zero of the destroyed page set, and *mask ldpnum* = *data ldpnum* = DESTROYED. Use of the Modification List when accessing and modifying the atomic page sets, and when committing or aborting transactions, is discussed below in Sections 3.7.2.2 and 3.7.2.3.

Another major data structure of the Atomic Page Set Subsystem, illustrated earlier in Figure 3-25, is the Commit B-Tree Buffer. It holds the most recently accessed pages of the Atomic Page Set Commit B-Tree, which is the data structure used to record the Commit List (the list of atomic page set modifications, made by transactions which are in the process of being committed). As with the Modification List, the Commit List is usually expected to be quite short. However, storing it as a B-tree will allow it to grow arbitrarily large, when needed. The structure of the Commit B-Tree is almost identical to that of the Modification B-Tree, except that the nodes of the tree are stored in dual (rather than normal) pages. See Figure 3-26 above for an illustration of the structure of a leaf node of the Commit B-Tree. Updating the Commit B-Tree must be done carefully (i.e. atomically with respect to system failures), just as in the case of updating the Permanent B-Tree. This is to avoid any possible corruption or accidental loss of transactions, after they have been flagged as "prepared to commit". A pointer to the root node of the Commit B-Tree is stored in page zero of the logical disk, so that the Atomic Page Set Subsystem can always find it at restart time.

The final major component of the Atomic Page Set Subsystem is the Data Page Buffer. Unlike the Standard Page Set Subsystem, the Atomic Page Set Subsystem must often manipulate subparts (byte ranges) within data pages. The relevant pages are held in the Data Page Buffer while they are being operated on. Similarly, modification Mask Pages are constructed, accessed, and modified by the Atomic Page Set Subsystem, using the Data Page Buffer for temporary storage.

3.7.2.2 Accessing and Modifying Atomic Page Sets

Any access or modification to an Atomic Page Set must be done in the context of a transaction, specified by *tid*. All modifications made in the course of a particular transaction are first recorded (temporarily) in the Modification List. For example, assume that the following invocation has been made:

---

[47] Although a bit map could be used in place of the current (byte map) Mask Page, the page oriented nature of the logical disk on which the maps are stored makes the current design more convenient. In addition, byte manipulations are usually more convenient and efficient than bit manipulations, in most current programming languages and hardware architectures.

nbw = **APSWrite**(tid, psid, location, nbytes, buffer)

Then the Modification List will be searched for (*tid*, *psid*, *pnum*), where *pnum* is the page containing the bytes specified by *location* and *nbytes*.[48] If not found, a new entry with that key will be inserted in the list, with pointers to newly allocated (and completely zeroed) Mask and Data Pages. In either case, the specified bytes of the Data Page are set to the contents of *buffer*, and the corresponding bytes of the Mask Page are "turned on" (set to hexadecimal FF).

Accessing the current contents of an atomic page set is a little more involved. For example, assume that the following invocation has been made:

nbr = **APSRead**(tid, psid, location, nbytes, buffer)

Then *tid* must be taken into account when determining which modifications (if any) have been made to the specified bytes, and should be visible within this transaction. The Modification List is first searched for (*tid*, *psid*, *pnum*), where *pnum* is the page containing the bytes specified by *location* and *nbytes*. The Modification List is then successively searched for the ancestor transactions of *tid*, to see if any of them have modified the page in question. As relevant entries are found, a Compound Mask and Compound Data Page are constructed, such that the modifications made in subtransactions take precedence over their ancestors. This can be easily accomplished (at least conceptually) using bitwise logical operations on entire Mask and Data Pages. For example, to merge another (ancestor) Data Page (*Data*) and Mask Page (*Mask*) into the Compound Data and Mask Pages that have been constructed thus far, the following equations can be used:

CompoundData = CompoundData **OR** (Data **AND** (**NOT** CompoundMask))
CompoundMask = CompoundMask **OR** Mask

If, at any point in the construction of the Compound Data and Compound Mask Pages, it is found that the entire page has been modified, then there is no need to search any further. The requested bytes can be obtained directly from the Compound Data Page, and returned in *buffer*. Otherwise, the modifications indicted by the Compound Mask and Compound Data Pages (if any) must be applied (conceptually) to the corresponding page in the Atomic Page Set Permanent B-Tree. This is done using the same formula as shown above, where in this case "Data" is the Permanent Data Page. The requested bytes can then be obtain directly from the Compound Data Page (as before), and returned in *buffer*.

Note that, because of the buffering provided in the Atomic Page Set Subsystem, most manipulations of the Modification List, including Mask and Data Page manipulations, will not require actual disk

---

[48] To simplify our examples, we will omit the details concerning the handling of multiple pages, and the crossing of page boundaries.

operations. Furthermore, page-at-a-time access to atomic page sets will usually be much more efficient than manipulating a small number of bytes with each operation, and it can be made even more efficient by treating it as a special case.

3.7.2.3 Transaction Commit and Abort Handling

All modifications to atomic page sets are first made on a temporary basis, by recording them in the Modification List, as outlined above. A modification will only become "permanent" after its associated transaction (*tid*) has been committed. In the case of a nested *elementary* subtransaction, a modification can only become permanent after the top level ancestor transaction commits. This is because any failure (or abort) of an ancestor transaction, will cause the modifications made in any elementary subtransactions to also be aborted. When a nested elementary subtransaction (*tid*) commits, all of its modifications become visible to its parent transaction (*parent-tid*). This is accomplished by simply "renaming" all of the *tid* entries in the Modification List, to have transaction ID *parent-tid*. In case of conflicts, the modifications made in *tid* take precedence over the modifications made in *parent-tid*. This is done by constructing a Compound Mask and Compound Data Page, using the same equations as discussed above.

Committing a top level elementary transaction, or any compound transaction (whether nested or not) causes all of its modifications to be permanently made to the affected atomic page sets. These modifications must be made very carefully, so as to preserve the atomic properties of the page sets (failure atomicity and durability). Since a transaction may span more than one logical disk, and more than one computing node, a two phase commit sequence is needed to coordinate the atomic updates of the various Atomic Page Set Subsystems, and to make them all occur as a single atomic update. In the first phase of the transaction commit sequence, the *APSPrepareCommit* primitive is invoked on each of the Atomic Page Set Subsystems involved in transaction *tid*. In response to *APSPrepareCommit*, each Atomic Page Set Subsystem must carefully record, in its Commit List, all of the modifications for transaction *tid*, so that they can no longer be lost in the event of a crash. This basically involves moving all of the entries with transaction ID *tid*, from the Modification List to the Commit List. However, to avoid inconsistencies in the Commit List, and the attendant loss of previously committed transactions, any updates to the Commit List must be done atomically with respect to system failures. The technique for atomically updating the Commit List B-tree is the same as that outlined below, for the case of the Atomic Page Set Permanent B-Tree.

After all of the Atomic Page Set Subsystems have indicated that they are prepared to commit transaction *tid*, the second phase of the commit sequence can begin. Phase two is signalled to each of the Atomic Page Set Subsystems by invoking the *APSCommit* primitive. In response to

APSCommit, each subsystem simply flags transaction *tid* as committed, and replies that it has completed the request. This allows the invoker to continue its execution, while the Atomic Page Set Subsystem actually carries out the required commit processing. Flagging a transaction as committed is accomplished by writing its transaction ID (*tid*) into the header portion of the root node of the Commit List B-Tree. Note that since the nodes of the B-tree are stored as dual logical disk pages, updating the root node on disk will be an atomic operation.

Once a transaction has been flagged as committed, an Atomic Page Set Subsystem will not accept any other requests for operations on atomic page sets, until it has completed the necessary commit processing. Committing transaction *tid* basically requires carefully (atomically) updating the "permanent" contents of the atomic page sets (as stored in the Atomic Page Set Permanent B-Tree), based on the modification information contained in the Commit List. The possible types of modifications are: (1) the insertion of new pages (including entirely new page sets); (2) the modification of existing pages; (3) the removal (zeroing) of existing pages; and (4) the removal (destruction) of entire page sets. The key to making all of the modifications for transaction *tid* occur atomically, is to never actually modify any existing node or data pages in the Permanent B-Tree (except one). Instead new pages, reflecting the required modifications, are constructed as necessary. Then, all of the modifications can be made permanent at the same time, by atomically updating a single Permanent B-tree node: the node at the root of the (minimum) subtree which encompasses all of the modifications. For more details concerning this technique, see [Mitchell 82], where the XDFS approach to atomically updating a file system B-tree is discussed.

After the Permanent B-Tree has been atomically updated, the modifications for transaction *tid* must be carefully removed from the Commit List, and *tid* erased from the header of the Commit List root node. This updating of the Commit List can all be done atomically, using the same technique as outlined above for the Permanent B-Tree. Note, however, that there is a period of time between the atomic updating of the Permanent B-Tree and the atomic updating of the Commit List, during which a system failure could occur. If a failure occurs during that time, the Atomic Page Set Subsystem will automatically (upon system restart) attempt to apply the modifications for transaction *tid* once again. This will not cause any problems, however, since all of the possible modifications to the Permanent B-Tree, as listed earlier, are *idempotent* (i.e. they can be repeated multiple times without changing the final result). When at last the Atomic Page Set Subsystem is able to complete the commit processing for transaction *tid*, it can then return to accepting new requests for operations on atomic page sets.

Besides committing, the other possible outcome for a transaction is that it aborts. A transaction (*tid*) can be aborted at any time, prior to being committed with *APSCommit*. The Atomic Page Set

Subsystem is notified of transaction aborts by means of the *APSAbort* primitive. When a transaction is aborted, all of the modifications made within it, or any of its nested subtransactions (except already committed compound subtransactions), must be removed (undone). This is accomplished by removing all entries for transaction *tid*, and all of its subtransactions, from both the Modification List and the Commit List. Note that if entries are to be removed from the Commit List, it must be done atomically, using the standard technique outlined above.

3.7.3 Logical Disk Subsystem

Each physical disk that is being used by the ArchOS system has a corresponding Logical Disk Subsystem Kernel Arobject. These Arobjects are created and destroyed as disks are mounted and dismounted (see Restart/Reconfiguration Subsystem, Section 3.7.4). A physical disk can have at most one ArchOS partition on it, and only that portion of the disk will be used by ArchOS. We refer to the ArchOS partition of a disk as a *logical disk*. A logical disk can simply be viewed as a sequence of pages, where a page is (currently) defined to be 2K bytes in size. Pages on the disk are numbered from 0 to N-1, where N is the total number of pages on the logical disk. A logical disk has a unique identifier permanently associated with it, regardless of where the disk is currently mounted.

Each instance of the Logical Disk Subsystem is responsible for managing the allocation of pages on its corresponding logical disk. It also keeps track of the bad (unusable) pages on the disk, so that they are not made available for allocation and later use. As much as possible, the Logical Disk Subsystem "optimizes" access to the physical disk, by allocating, reading, and writing multiple pages at a time, and ordering operations so as to minimize disk head movement. It also provides a degree of "network transparency", by automatically buffering and transferring data between the local and remote machines.

In addition to NORMAL (single) pages, the Logical Disk Subsystem also supports the concept of DUAL pages. A DUAL page is written more carefully than a NORMAL page, so that if a crash occurs while writing a DUAL page, its original contents can still be recovered. In this way, a simple, basic form of failure atomicity is provided. As the name suggests, each DUAL page is implemented using two single pages. DUAL pages are used for saving important data structures, and as a building block for implementing more complex atomic operations.

The Logical Disk Subsystem provides the following set of primitives:

```
pagelist = DKAllocate(npages, type [, follows])
val = DKFree(pagelist)

npr = DKRead(pagelist, buffer)
npw = DKWrite(pagelist, buffer)

val = DKSync()
val = DKStatus(statusbuffer)

val = DKRestart(dev-name)
val = DKRecoverDual(pagelist)

val = DKUnmark()
val = DKMark(pagelist)
val = DKGood()
val = DKBad(pagelist)
```

PAGELIST pagelist List of 32-bit logical disk page numbers, where the first 8 bits of a page number are used for various flags, and the remaining 24 bits are converted to the cylinder, track, and sector numbers of the physical disk. The first flag bit indicates whether the page is DUAL or NORMAL.

BOOLEAN val TRUE if the specified operation is done successfully; otherwise FALSE.

INT npr The actual number of pages read.

INT npw The actual number of pages written.

INT npages The number of pages to be allocated.

PAGETYPE type The type of pages to be allocated: NORMAL or DUAL.

PAGEID follows Logical disk page number, following which the new pages are to be allocated (physically as close as possible).

BUFFER *buffer The buffer address in the kernel address space of either the local or remote node.

DKSTATUSBUFFER *statusbuffer
The buffer address for returning status information (can be either a local or remote kernel address).

DEVNAME dev-name
The logical device name.

The *DKAllocate* primitive allows a number of pages to be allocated at a time, and it returns a list of the allocated logical disk page numbers. An attempt is made to allocate pages which are physically close together on disk. The pages can be of type NORMAL or DUAL, and a preferred location on disk (near an existing page) can be specified. The *DKFree* primitive deallocates the specified list of pages.

DKRead reads a list of pages into the specified kernel buffer, and returns the count of pages actually read. Similarly, *DKWrite* writes a list of pages from the specified kernel buffer, and returns the count of pages actually written. When the buffer is in a remote kernel, *DKRead* and *DKWrite* automatically handle the cross node copying of data.

The *DKSync* primitive ensures that any buffered information within the Logical Disk Subsystem is consistent (synchronized) with the version on disk. The *DKStatus* primitive provides information about disk utilization, frequency of access, frequency of errors, number of bad blocks, and so on. It can be used, for example, by the Arobject/Process Management Subsystem, to determine the placement of process address space paging areas.

DKRestart is used only when the system is restarted. It initializes the disk hardware and any internal data structures. The *DKRecoverDual* primitive recovers the given list of DUAL pages, by ensuring that the two copies of each DUAL page are consistent (see [Sturgis 80] for details). This is used by various subsystems when recovering their data structures after a crash.

The *DKUnmark* and *DKMark* primitives are provided to help in garbage collection. First, *DKUnmark* is used to flag all of the pages on the disk (except bad pages) as free. Then, *DKMark* is used (repeatedly) to flag the specified pages as allocated. Similarly, *DKGood* and *DKBad* are used for managing the bad pages (unusable pages) on the disk. *DKGood* flags all of the pages on the disk as good. Then, *DKBad* is used to flag pages that have been determined to be bad.

3.7.3.1 Components of the Logical Disk Subsystem

The Logical Disk Subsystem consists of a single Manager process, and three main data structures, which are shown in Figure 3-27. The Logical Disk Manager (LDM) process accepts all operations for the subsystem, executes them, and returns the results. The three data structures it uses are: (1)Sorted Page List, (2)Data Buffer, and (3)Page Allocation Map buffer.

The purpose of the *Sorted Page List* is to order the disk pages to be read or written, so as to reduce disk head movement, and improve disk performance. Each entry of the Sorted Page List consists of three values: the logical page ID of the starting page, the number of pages (*npages*) to be read or

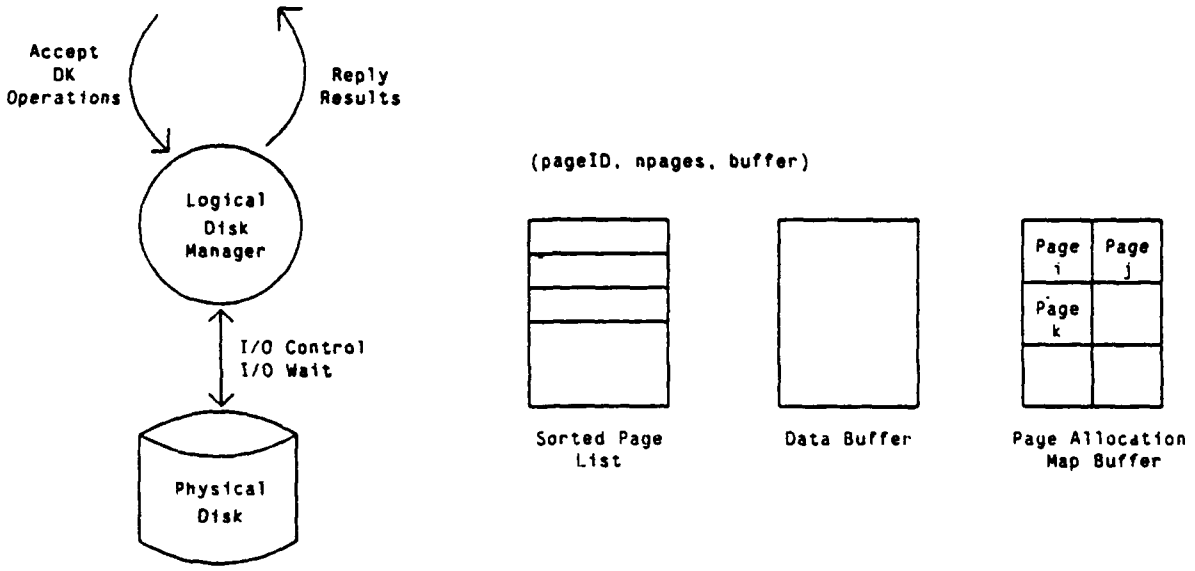


Figure 3-27: Components of the Logical Disk Subsystem

written in one disk operation, and the location within the kernel buffer to be used for the operation. The Logical Disk Subsystem allows multiple page read and write operations. The *pagelist* provided is clustered (if possible) and then entered in the Sorted Page List.

The Data Buffer provided is used only for remote operations (invoked from other nodes in the distributed system). For example, data read from the disk (for a remote operation) is first placed in the Data Buffer, and then transferred to the destination buffer on the remote node, by using the system *copy* facility. A "double buffering" technique is used, so that the disk transfer and *copy* operations can proceed in parallel.

The Page Allocation Map Buffer is used to buffer some of the pages of the Page Allocation Map. Whenever the Page Allocation Map has to be read or modified (such as in *DKAllocate*, *DKFree*, *DKMark*, and *DKUnmark*), the relevant pages of the map are first read into the buffer (unless they are already buffered). A detailed description of the Page Allocation Map and its handling is provided in Section 3.7.3.3.

3.7.3.2 Disk Layout

A logical disk is laid out as a sequence of pages, which are numbered from 0 to N-1, where N is the total number of pages on the logical disk. The *logical page number* specified in the logical disk operations consists of the concatenation of a *flags* byte, with a number from 0 to N-1. A page can either be NORMAL or DUAL. A NORMAL page is a single disk page, referred by a logical page number. A DUAL page consists of two pages, where the second page is located a fixed offset from the first (primary) page. The DUAL page is referred to by the logical page number of the primary page. The first bit in the *flags* byte (of the logical page number) indicates whether a page is NORMAL or DUAL. The two physical pages of a DUAL page have a gap of one page (pages are not adjacent), to reduce the probability of both pages being damaged by a single disk fault, and to ensure reasonably efficient sequential access.

Logical page 0 of the disk serves a special function. It contains information about the logical disk, and pointers to all important data structures saved on the disk. Thus, it saves information about the disk drive characteristics, the logical disk name, the logical disk size, and pointers to the Page Allocation Map, the Bad Page Table, the File System Directory, and so on. For reliability, page 0 is a DUAL page.

3.7.3.3 Disk Page Allocation

The Page Allocation Map is a bit map, which has one bit corresponding to each page on the logical disk. If a disk page has been allocated, its bit in the allocation map is set. The Page Allocation Map itself is saved near the middle of the disk, to provide efficient access. All pages of the allocation map are DUAL, and laid out as shown in Figure 3-28.[49]

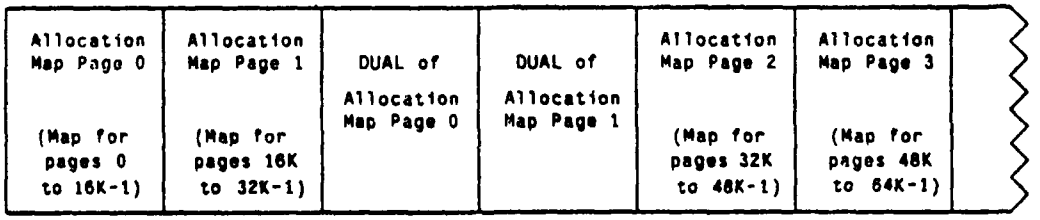


Figure 3-28: Layout of the Page Allocation Map

[49] If we have a 500 MB logical disk, it would contain 250K pages of size 2K. The size of the allocation map would be 250K bits, or 16 DUAL pages (32 logical disk pages) approximately.

The page allocation algorithm used is fairly simple. It attempts to pack the allocated pages, starting at page 1 of the disk, and continuing through to page N-1. If the last page allocated was page *i*, the following allocation will start from page *i + 1* (unless a preferred allocation page is specified). The advantage of this scheme is that allocation and deallocation are very efficient. For allocation, the page of the allocation bitmap containing bit *i + 1* is likely to already be buffered, since that page was probably used recently (for the previous allocation operation). Hence, no disk reads will usually be required for allocation. For deallocation, the appropriate allocation map page can be easily determined, then read in (if necessary), and modified. Note that with this allocation scheme, successively allocated pages will tend to be located close together on disk, thus improving access efficiency.[50]

3.7.3.4 Bad Page Handling

One of the functions of logical disk management is to prevent the use of bad pages on the disk. A list, called the Bad Page List, is maintained near the end of the disk, and consists of the logical page numbers of all known, unusable (bad) pages.[51] The Bad Page List is stored as DUAL pages. When the Page Allocation Map is initialized (using *DKUnmark*), all the bad pages are marked as allocated, to prevent further usage of these pages.

The Bad Page List is usually constructed when a new disk is initialized, and is not expected to change very much after that. When the Bad Page List is constructed, the Data Buffer can be used for buffering it. The List will be constructed by a special utility, which first calls *DKGood* to remove the old Bad Page List (if it exists). After that, the entire disk is scanned to find any bad pages. This is done by writing each page and then reading it back. For each bad page found, its logical page ID is entered in the Bad Page List (with *DKBad*). Once the system is in operation, if a bad page is discovered by some subsystem, it can call *DKBad* to enter the page in the Bad Page List.

---

[50] This allocation scheme will be able to cluster multiple page allocations reasonably well when the disk is new, but the disk will slowly become more and more fragmented with continued use. Some improvements could be made, to reduce the amount of fragmentation, if it is found to have a significant impact on performance.

[51] The entries in the Bad Page List are sorted when the list is first constructed, but new entries are simply added to the end of the list as found. The Bad Page List is expected to require only a few disk pages for storage. If 1% of the pages (size 2KB each) of a 500 MB disk are bad, the Bad Block List would have 2.5K entries. Given that each entry is 4 bytes long, 5 DUAL pages (10 disk pages) are required to save the entire list.

3.7.4 Restart/Reconfiguration Subsystem

The Page Set Restart/Reconfiguration Subsystem (PSR Subsystem) performs several functions related to the restart of a computation node following a crash, as well as the mounting and unmounting of logical disks. A single instance of this subsystem resides at each node in the system. The PSR subsystem "bootstraps" the entire Page Set Subsystem at restart time. After a crash, the Restart/Reconfiguration Subsystem is restarted automatically, and is responsible for constructing the entire Page Set Subsystem. The kernel (on a node) maintains a list of device addresses which it probes, to determine the logical disks on its own computing node. The PSR Subsystem sets up the atomic and standard page set arobjects, as well as the logical device arobject, for each logical disk found on the node.

In addition to restarting the Page Set Subsystem, the PSR Subsystem adds and removes logical disks dynamically (while the system is running) as necessary. When a new logical disk is added, the various Page Set Subsystem arobjects (Standard Page Set, Atomic Page Set, and Logical Disk) have to be created for it. When a disk is removed, the corresponding arobjects have to be killed. Some support is provided for the clean unmounting of disks. It is possible to quieten activity on a disk (by allowing old activity to complete, but refusing new activity), prior to disk removal.

The PSR Subsystem maintains a global Mount Table, which indicates the location (node) of all the logical disks in the entire distributed system, and the Page Set Subsystem Arobjects associated with each logical disk. Each instance of the PSR Subsystem maintains a copy of the Mount Table, and the peers co-operate in keeping up-to-date versions of the table. The Mount Table at a node is globally accessible to other kernel level subsystems on that node. Along with the global Mount Table, a local Unmount Table is also maintained. The Unmount Table lists the logical disks accessible to the PSR Subsystem, but not globally visible to other subsystems on its node, or to other nodes in the distributed system.

The PSR Subsystem performs two other useful functions. Firstly, it co-ordinates garbage collection for the entire Page Set Subsystem (on a node). Secondly, it checks any given disk for bad (unusable) pages, and enters them in a Bad Page List (refer Section 3.7.3). Each of these two functions can be performed at restart, as well as when the system is running.

The PSR Subsystem allows a logical disk to be in one of four possible states: Mounted, Unmounted, Quiet, or Removed (refer Figure 3-29). In the *Mounted* state, the logical disk has an entry in the Mount Table, and is globally visible on its own node, as well as throughout the system. All types of disk operations (except disk checking and garbage collection), such as paging, reading and writing, are

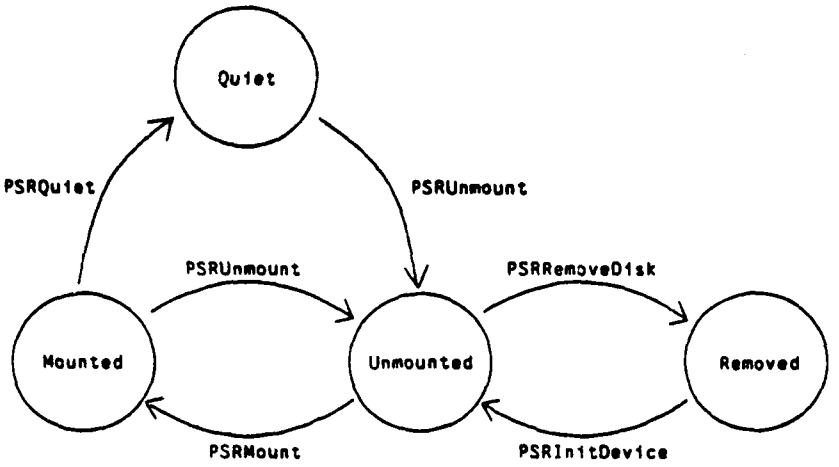


Figure 3-29: State Diagram for Logical Disks

permitted in this state. The *Quiet* state is very similar to the *Mounted* state in having an entry in the Mount Table, and being globally visible. The only difference is that this state represents an attempt to quiet down system activity prior to disk removal, and hence new activity is not allowed. In the *Unmounted* state, the logical disk has an entry only in the Unmount Table, and is visible within the PSR Subsystem. The main purpose of this state is to allow disk checking, garbage collection, and other disk maintenance to be performed, while the rest of the system is still up and running. The *Removed* state corresponds to the physical removal of the disk from the computer system.

val = **PSRRestart**(device-list [, options])
diskid = **PSRInitDevice**(dev-name [, initflags])
val = **PSRRemoveDisk**(diskid)
val = **PSRMount**(diskid)
val = **PSRUnmount**(diskid)
val = **PSRQuiet**(diskid)

val = **PSRCheckDisk**(diskid)
val = **PSRGarbageCollect**(diskid)

val = **PSRInsertMount**(diskid, node, disk-aid, ps-aid, aps-aid, mtflags)
val = **PSRRemoveMount**(diskid)
val = **PSRRequestMount**(mount-table)

BOOLEAN val TRUE if the specified operation is done successfully; otherwise FALSE.

DISKID diskid The Logical Disk ID of the disk volume being operated on.

DEVLIST device-list
The list of logical device names corresponding to logical disks in the system.

PSROPTIONS options
A set of boolean flags which can be specified as restart options for operations to be carried out in the restart sequence. The flags provided are: GARBAGE-COLLECT, and CHECKDISK.

DEVNAME dev-name
The logical device name.

PSRINITFLAGS initflags
A set of boolean flags which can be specified when a logical disk is initialized. Currently, two flags are provided: READ-ONLY, and SELF-CONTAINED.

NODENAME node The ID of the node on which the disk is mounted.

AID disk-aid The arobject ID for the Logical Disk Subsystem Arobject corresponding to the disk *diskid*.

AID ps-aid The arobject ID for the Standard Page Set Subsystem Arobject corresponding to the disk *diskid*.

AID aps-aid The arobject ID for the Atomic Page Set Subsystem Arobject corresponding to the disk *diskid*.

PSRMOUNTFLAGS mtflags
A set of boolean flags associated with each entry in the Mount Table. The flags provided are: QUIET, READ-ONLY, SELF-CONTAINED.

MOUNTBUFFER *mount-table
The buffer used for transferring the contents of the mount table between different nodes in the system.

The *PSRRestart* primitive restarts the Page Set Subsystem on a given node, following a node crash. It is invoked by a kernel process responsible for bringing up the entire kernel on that node. The list of devices corresponding to all logical disks on the node is determined by the kernel process, and supplied as a *PSRRestart* parameter. For each logical disk on the node, the *PSRRestart* primitive creates and restarts three page set subsystem arobjects (standard page set, atomic page set, and logical disk). In addition, if the GARBAGE-COLLECT and/or CHECKDISK options are specified, then garbage collection and/or disk checking (for bad pages) are performed as part of the page set restart sequence.

The *PSRInitDevice* primitive adds a new logical disk to the node. The state of the logical disk makes a trasition from the *Removed* state to the *Unmounted* state. The system device name (*dev-name*) for the logical disk is supplied, and its logical ID is found (from page 0 of the disk) and returned. The Standard Page Set, Atomic Page Set, and Logical Disk arobjects are created for the newly added disk. Optional flags (READ-ONLY, and SELF-CONTAINED) can be specified as necessary, for restricting the type of disk access permitted. The READ-ONLY flag specifies that the entire disk is read-only, and its contents cannot be modified. The SELF-CONTAINED flag specifies that all files saved on that disk must have the directory entries, as well as the file contents (page sets) saved on that disk.[52] The *PSRRemoveDisk* primitive removes a specified *unmounted* logical disk from the system. The three page set arobjects for the logical disk are destroyed.

The *PSRMount* primitive mounts the specified unmounted logical disk, and makes it globally accessible. All nodes in the distributed system are informed about the newly mounted disk. The *PSRUnmount* primitive unmounts the specified logical disk, from the *Mounted* or *Quiet* state. In either case, the result of a *PSRUnmount* is seen by the rest of the system as a disk crash, where no further operations are permitted, and ongoing operations cannot be completed.[53] It is expected, however, that if the disk was *quiet* prior to unmounting, there would be very little (if any) ongoing activity at this time. The *PSRQuiet* primitive flags a mounted logical disk as being in a special *quiet* state, in which ongoing activity can be continued, but new activity is not allowed. The main prupose of this primitive is to allow for clean disk unmounting.

The *PSRCheckDisk* and *PSRGarbageCollect* primitives are allowed only when the specified logical disk is in the *Unmounted* state. The *PSRCheckDisk* primitive determines the bad pages on the specified disk (non-destructively), and constructs the Bad Page List. The *PSRGarbageCollect* primitive co-ordinates garbage collection for the specified disk, by invoking garbage collection primitives provided by the standard page set, atomic page set, and logical disk arobjects.

The Mount Table is replicated on all nodes in the distributed system. Three primitives are provided, which are used only by peer Restart/Reconfiguration arobjects on other nodes, for obtaining and updating information from the Mount Table. The *PSRInsertMount* primitive is used for adding a new Mount Table entry, or for modifying an existing entry. The parameters specified are the logical disk ID (*diskid*), the ID of the node on which the disk is mounted (*node*), the IDs of the three Page Set

---

[52] If a disk is self contained, it can be easily moved from one ArchOS system, and initialized on another.

[53] *PSRUnmount* can be implemented as the destruction of the three page set arobjects (standard page set, atomic page set, and logical disk), followed by the creation of new page set arobjects. In this case, all requests for operations with the old page set arobject IDs will return error indications.

Subsystem arobjects: logical disk subsystem (*disk-aid*), standard page set (*ps-aid*), and atomic page set (*aps-aid*), and a set of flags. The *PSRRemoveMount* primitive removes the Mount Table entry for the logical disk. Finally, the *PSRRequestMount* primitive returns the contents of the entire Mount Table in the buffer provided. This primitive is primarily invoked by a peer PSR arobject which is restarting after a crash, and attempting to reconstuct its copy of the system-wide Mount Table.

3.7.4.1 Components of the Restart/Reconfiguration Subsystem

The PSR Subsystem consists of four types of components: (1)the Mount Table, (2)the Unmount Table, (3)the PSR Manager process, and (4)one or more PSR Worker processes. Each of these components is briefly described below, and shown in Figure 3-30.

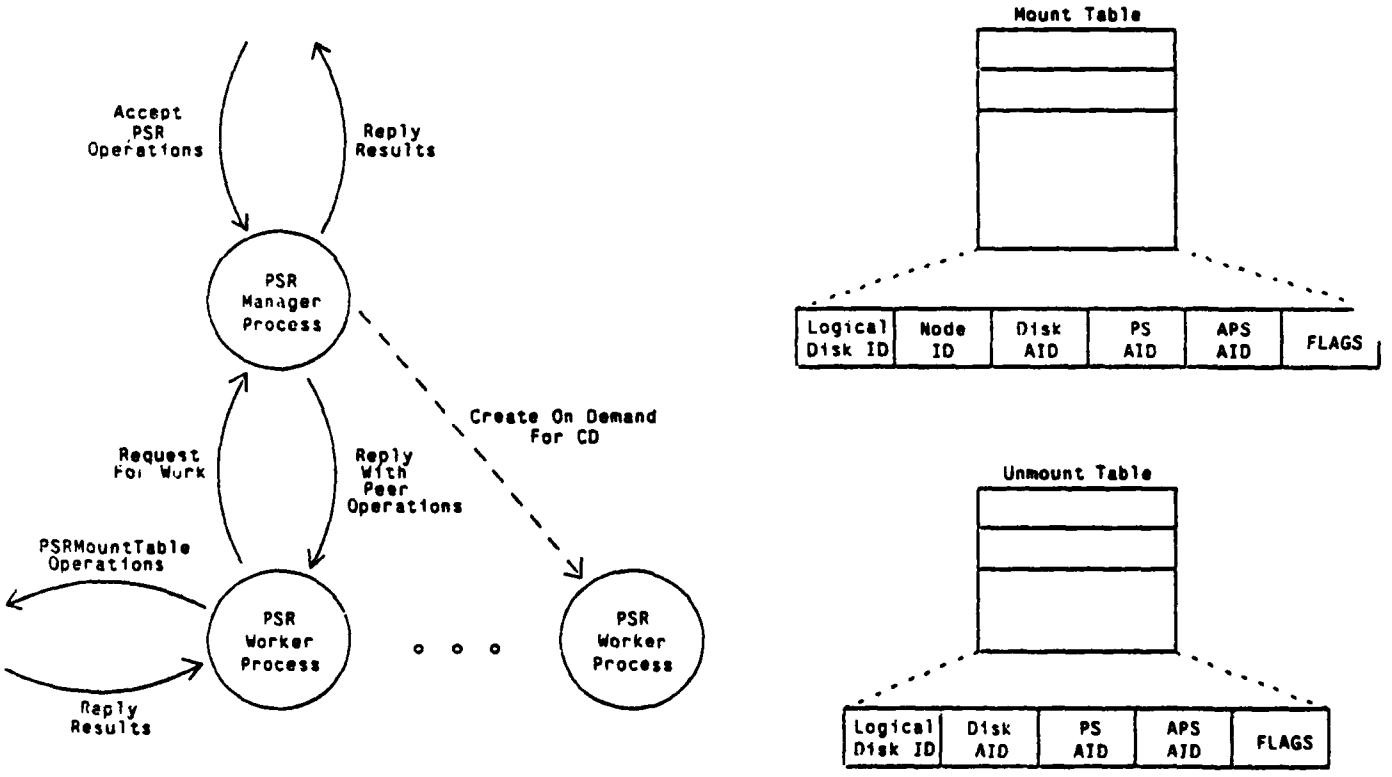


Figure 3-30: Components of the Restart/Reconfiguration Subsystem

The Mount Table provides information about the location of all (*mounted* and *quiet*) logical disks in the entire distributed system, and the arobjects which are responsible for managing these disks. A copy of the Mount Table is maintained at each node in the system. The Mount Table is globally visible

to all subsystems on a node (and to other nodes), but is updated only by the PSR Subsystem. During updates, the table is locked, to prevent the access of inconsistent states of the table. Each entry of the Mount Table contains the following items of information: the ID of the logical (*mounted* or *quiet* disk, the ID of the node on which it is mounted, the ID of the Logical Disk Subsystem arobject for that disk, the ID of the Standard Page Set arobject for that disk, the ID of the Atomic Page Set Arobject for that disk, and a set of flags (QUIET, READ-ONLY, and SELF-CONTAINED). The QUIET flag is the only way of specifying whether a logical disk is in the *mounted* state, or in the *quiet* state. The READ-ONLY and SELF-CONTAINED flags specify particular types of disk usage, and has been discussed in the previous section.

The Unmount Table provides information about *unmounted* logical disks, which can be accessed by the PSR Subsystem, but are not visible to the rest of the system. It is useful to be able to access disks in this mode, primarily for maintenance purposes, such as disk checking and garbage collection. The entries in the Unmount Table are very similar to those in the Mount Table, except that the ID of the node is not required, since the Unmount Table refers only to its local node. In the flags field, the QUIET flag is not required for this table.

The PSR Manager process is responsible for providing most of the functionality of the PSR Subsystem. It accepts all PSR operations, and returns the results. It also manages the Mount and the Unmount tables. The PSR Worker processes serve two functions. They implement *PSRCheckDisk* and *PSRGarbageCollect* operations on behalf of the Manager. Since both these operations can potentially take a very long time, the PSR Subsystem can still be used for performing operations on other disks concurrently. The other function of the Worker process is to wait on behalf of the Manager process, when peer level operations are invoked by the Manager. This prevents deadlocks arising from multiple, concurrent peer operations.[54] One Worker process is always present in the PSR Subsystem for invoking peer PSR operations (PSRInsertMount, PSRRemoveMount, and PSRRequestMount). For each *PSRcheckDisk* or *PSRGarbageCollect* operation, a new Worker process is created, and then destroyed when the operation is completed.

3.7.4.2 Restart

The PSR Subsystem performs two related functions when restarting after a node crash. The Mount Table has to be recreated, and the Page Set Subsystem has to be brought up. The reconstruction of the Mount Table is quite similar to the reconstruction of the Directory Map Table, when the Directory Map Subsystem of the File System is restarted (refer Section 3.6.3). The PSR Manager process

---

[54] Note the similarity with the handling of the Directory Map Table in the File Subsystem in Section 3.6.3.

invokes a Worker process to obtain the Mount Table (with *PSRRequestMount*) from one of the other nodes in the system. In the meantime, it proceeds to initialize all the logical disks on its own node, which have been found by the kernel restart process. It determines the IDs of the logical disks, and mounts them.

For each logical disk on the node, its ID is first determined, and an entry made in the Unmount Table. Next, a Logical Disk Subsystem arobject is created for that disk, and then restarted. As a result of the Disk Subsystem restart, all the logical disk data structures are recovered (page 0 of the disk, the allocation map, and the bad page list). Once this operation has completed, the standard page set arobject is created and restarted, so that the Page Set B-tree is recovered correctly. Next, the atomic page set arobject is created and restarted. This allows all its data structures to be recovered, and all committed transactions to be completed. The IDs for the three newly created arobjects are entered in the Unmount Table. Next, the logical disk is *mounted*, and entered in the Mount Table. Once all the logical disks are thus mounted, all the peer PSR arobjects are told to update their Mount Tables, using the *PSRInsertMount* primitive.

If the disk checking and garbage collection options are specified, then these operations also have to be performed as part of the restart sequence. Garbage collection is a good idea after a restart, so that all temporary page sets can be flushed.

3.7.4.3 Garbage Collection and Disk Checking

The PSR Subsystem co-ordinates garbage collection for the Page Set Subsystem. It first calls on the *DKUnmark* primitive, so that the Logical Disk Subsystem marks the entire allocation map as being free. Next, it calls the *PSGarbageCollect* primitive, which specifies all the pages being used by the Standard Page Set Subsystem. Following that, the *APSGarbageCollect* primitive is invoked, which marks all the disk pages in use by the Atomic Page Set Subsystem. As a result of this garbage collection procedure, all temporary page sets, as well as pages not in use by the Page Set Subsystem are freed.[55]

Disk checking determines all the unusable (bad) pages on the specified disk in a non-destructive manner. First, the *DKGood* primitive is called to remove the old Bad Page List, thereby marking all disk pages as "good". Then each page on the disk is read. If there is no checksum error in reading, then the page is usable. If there is an error, the page is written into, and then read again. If the error persists, then the page number is entered in the Bad Page List. All the pages of the disk are checked in this way.

---

[55] It is still possible for unreferenced files to exist, since this garbage collection procedure does not consider subsystems above the Page Set Subsystem.

3.8 I/O Device Subsystem

An I/O device is treated as a special type of file where no transaction is allowed and a device specific control scheme is included. To read, write or issue a special command such as reset, a device must be opened before any read/write access and it has to be closed after all of the actions are completed.

All of the device dependent commands can be sent to devices by using the SetIoctl primitive.

```
dd = OpenDevice(devicename, mode)
CloseDevice(dd)

nr = ReadDevice(dd, buf, nbytes)
nw = WriteDevice(dd, buf, nbytes)

event-cnt = Iowait(dev-descriptor, timeout)

val = SetIoctl(dev-descriptor, io-command, dev-buf, timeout)
```

It should be noted that the current I/O system does not interact with the transaction manager at all. Thus, any type of transaction facility is provided.

3.9 Time-Driven Scheduler Subsystem

Process scheduling in a real-time facility is crucial to the success of the system. Process scheduling in ArchOS differs fundamentally from scheduling on other operating systems because of several critical factors:

- ArchOS is a real-time system; hence process scheduling must be compatible with its critical goal of supporting application-defined time constraints
- ArchOS design manages application time-constraints by explicitly accounting for the fact that there is a definable time-varying value to the system for completing each process at a particular time.

3.9.1 Best-Effort Scheduling

3.9.1.1 Value Function Processing

The ArchOS scheduler, part of the ArchOS Kernel Arobject (see Section 3.3.1 designed to explicitly use the application defined value function for the set of processes in its queue in making a "best-effort" decision on the process to be executed at each point in time.

The computational model for this scheduler consists of a set of schedulable processes p_i resident in a processor. Each such process has a request time R_i, an estimated computation interval C_i and a value function $V_i(t)$, where t is a time for which the value is to be determined. Figure 3-31 illustrates these process attributes for a process with a linearly decreasing value function prior to a critical time D_i, and an exponential value decay following the critical time. The illustration depicts a process which is dispatched after its request time and which completes prior to its critical time without being preempted.

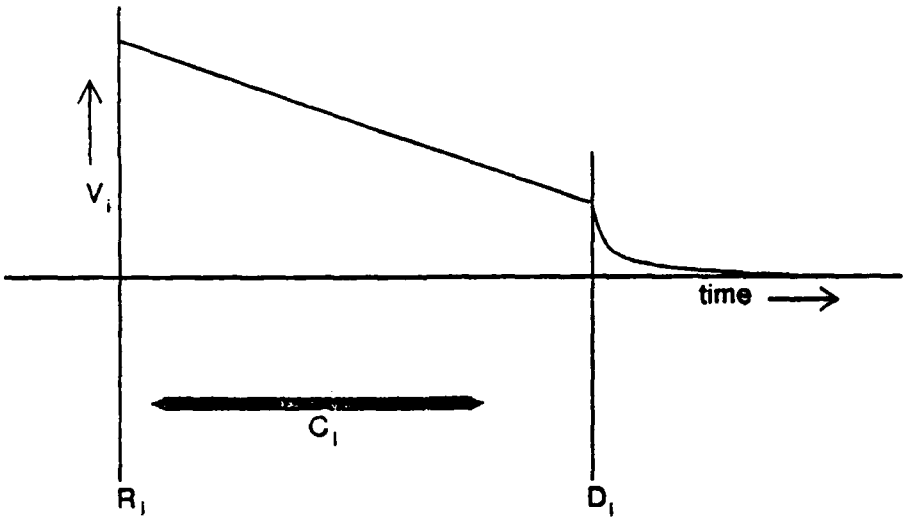


Figure 3-31: Process Model Attributes for Process i

$V_i(t)$ defines the value to the system due to completing p_i at time t. The set of these value functions is used by the scheduler to determine the best sequence in which to schedule each of the available processes. The type of functions definable for V_i will determine the range of scheduling policies supportable by the operating system, particularly with respect to the handling of a processor overload in which some critical times cannot be met.

We note that the existence and importance of the deadline is dependent on the value function. The value function can be said to define an explicit deadline only if it has a discontinuity in the function, its first derivative, or its second derivative, in which the value is lower or decreasing after the

discontinuity. In Figure 3-31, the deadline is defined by the discontinuity in its first derivative at the critical time D_i.

At any particular point in time, there will be n processes ready for scheduling, resulting in n! possible scheduling sequences. Each of these sequences consists of a process ordering $(m_1, \cdots, m_n)$, where p_{m_j} would be the j^{th} process to be scheduled. A scheduling sequence will be considered optimal if, with respect to the available information at the time of the scheduling decision, β is maximized, where $\beta = \sum V_i(T_i)$ and T_i is the actual completion time of p_i using this scheduling sequence (i.e., if p_i is the j^{th} process to be scheduled, then $T_i = \sum_{k=1}^{j} C_{m_k}$). See Figure 3-32 for an example of four processes with value functions, for which the choice of a best effort schedule is non-trivial. The figure shows the four value functions, and a potential scheduling sequence such that each completes with a high value.

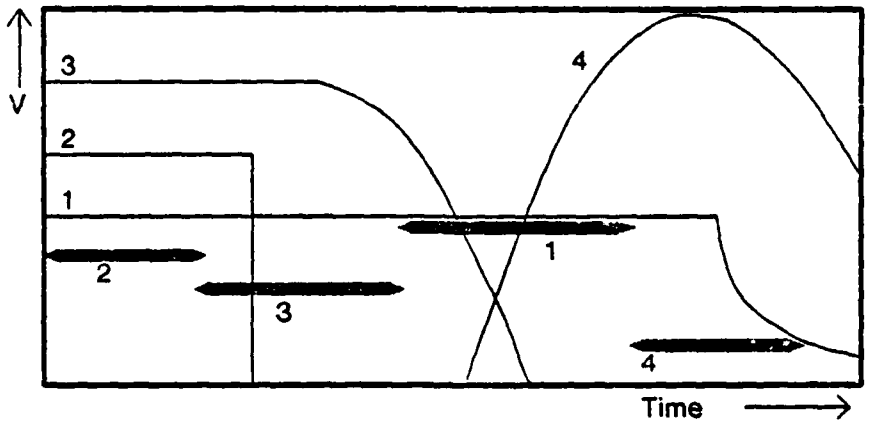


Figure 3-32: Four "Typical" Processes with their Value Functions

Since the completion times used for this scheduler will be known only stochastically, the assumed distribution and its computed parameters (e.g., mean, variance) will be used to compute its expected value in the scheduling sequence, resulting in a statistically "good" sequence. Making an optimum schedule can be shown to be computationally intractable, so the best-effort scheduler will use a set of heuristics to determine a sequence which will generate a high total value over any time period which is sufficiently long with respect to the completion times of the processes to be scheduled.

3.9.1.2 Well-Known Scheduling Algorithms

Scheduling a set of processes consists of producing a sequence of processes on one or more processors such that the utilization of resources optimizes some scheduling criterion. Criteria which have historically been used to generate process schedules includes maximizing process flow (i.e., minimizing the elapsed time for the entire sequence), or minimizing the maximum lateness (lateness is defined to be the difference between the time a process is completed and its deadline). It has long been known [Conway 67] that there are simple algorithms which will optimize certain such criteria under certain conditions, but algorithms optimizing most of the interesting scheduling criteria are known to be NP-complete [Garey 79], indicating that there is no known efficient algorithm which can produce an optimum sequence. Clearly, the choice of metric is crucial to the generation of a processing sequence which will meet the goals of the system for which the schedule is being prepared.

There are several well-known scheduling algorithms which have traditionally been used in process scheduling, each with properties which make it useful for certain applications. Among these are:

1. **SPT**. At each decision point, the process with the shortest estimated completion time is executed. This algorithm maximizes overall throughput, and is frequently used (although in modified form) in batch systems.

2. **Deadline**. At each decision point, the process with the earliest deadline is executed. This algorithm, in the absence of an overload, will result in meeting all deadlines. More precisely, this algorithm will minimize the maximum lateness.

3. **Slack**. At each decision point, the process with the smallest estimated slack time (elapsed time to the critical time minus its estimated completion time) is executed. This algorithm, in the absence of an overload, will also result in meeting all deadlines, but will produce a much higher level of preemptions. This algorithm will maximize the minimum lateness.

4. **FIFO**. At each decision point, the process which has been in the request set longest is executed. This algorithm will produce a relatively "fair" schedule, in which lateness will be spread out to all processes.

5. **Priority**. At each decision point, the process with the highest fixed priority is executed. The "most important" process is executing at any moment.

Of these, real-time systems traditionally use only the Priority and/or FIFO schedulers. It should be noted that no objective performance measures (e.g., meeting deadlines, high throughput, low lateness or tardiness) are even approximately optimized by these algorithms, but they are inexpensive to implement and require very few run-time resources.

In priority-driven scheduler systems, deadline management is attempted by assigning a high fixed priority to processes with "important" deadlines, disregarding the resulting impact to less "important" deadlines. During the testing period, these priorities are (usually manually) adjusted until the system implementer is convinced that the system "works". This approach can work only for relatively simple systems, since the fixed priorities do not reflect any time-varying value of the computations with respect to the problem being solved, nor do they reflect fact that there are many schedulable sets of process deadlines which cannot be met with fixed priorities. In addition, implementers of such systems find that it is extremely difficult to determine reasonable priorities, since, typically, each individual subsystem implementer feels that his or her program is of high importance to the system. This problem is usually "solved" by deferring final priority determination to the system test phase of implementation, so the resulting performance problems remain hidden until it is too late to consider the most effective design solutions.

Using a deadline scheduler(i.e., a scheduler which schedules the process with the closest deadline first [Liu 73]) solves the problem of missing otherwise schedulable deadlines due to the imposition of fixed priorities, but leaves other problems, most notably the problem of the transient overload. The deadline scheduler provides no reasonable control of the choice of which deadlines must be delayed in an overload, leading to unpredictable failures and resulting in an impact on reliability and maintainability.

3.9.1.3 A Best-Effort Scheduling Algorithm

The creation of a best-effort scheduling algorithm is one of the key research interests of the Archons project, and work on it is currently underway at this writing. However, there are some preliminary results which have been produced, and which will be used, at least in our initial scheduler design. In this initial algorithm, we take advantage of several observed value function and scheduling characteristics:

- Given a set of processes with deadlines *which can all be met* based on the sequence of the deadlines and the computation times of the processes, it can be shown that a schedule in which the process with the earliest deadline is scheduled first (i.e., a *Deadline* schedule) will always result in meeting all deadlines.

- Given a set of processes (ignoring deadlines) with known values for completing them, it can be shown that a schedule in which the process with the highest value density (*V/C*, in which V) is its value and C) is its processing time) is processed first will produce a total value at any period of time no lower than any other schedule.

- Most value functions of interest (at least among those investigated at this time) have their highest value occurring immediately prior to the critical time.

Some of the implications of these observations are:

- If no overload occurs, all deadlines will be met, and the value function produced by the deadline schedule will be as high as possible.
- If an overload occurs, and some processes must miss their deadlines, the Value Density Schedule would produce a high value.
- Therefore, if we can predict the probability of an overload, we can choose processes with low value density as candidates for being removed from a deadline schedule, until a deadline schedule is produced which has an acceptably low probability of producing an overload.

The algorithm we will use, then, will start with a deadline-ordered sequence of the available processes, which will be sequentially checked for its probability of overload. At any point in the sequence in which the overload probability passes a preset point, the process prior to the overload with the lowest value density will be removed from the sequence, repeating until the overload probability is acceptable.

3.9.2 Time Management

The ArchOS scheduler will use information provided by the time management primitives to make its scheduling decisions. This information is derived from these primitives:

---

rtc = **GetRealTime**()
Timedate(time, date)

TimeDate = **Delay**(deltime, criticaltime, comptime, setflag)
TimeDate = **Alarm**(alarmtime, criticaltime, comptime, setflag)

val = **SetTimeDate**(time, date)

In addition to these primitives, policy directives will be used which are provided by the Policy module (see Section 3.2.2.3).

The information needed for making the scheduling decisions includes:

1. **The request time.** This is the time a process becomes available for execution, although its value function may be negative, forcing the scheduler to delay it until it can be completed with as high a positive value as possible. Its value function is defined by the process itself using the **delay** or **alarm** primitives.
2. **The critical time.** This time is the time relative to the request time for the process at which the value function may have a discontinuity, and is determined when the process

was requested. Its value relative to the request time is set by the process itself using the **delay** or **alarm** primitives.

3. **The value function parameters themselves.** The value function is divided into two parts: (1) the value for completing the process prior to its critical time, and (2) the value for completing the process after its critical time. In each of these functions, the time used in computing the function is relative to the critical time, so the value function used prior to the critical time measures its time "backwards". Each function has the form: $V(t = K_1 + K_2t - K_3t^2 + K_4e^{-K_5t})$ so there are ten parameters defining the total value function. Value function parameters are defined by the application programmer, but are modified by the system policy currently in effect. In the case of processes (i.e., server processes) scheduled on behalf of other processes (i.e., client processes), the value function can, at the option of the process, use either the value function of its client or its own value function.

4. **System policy.** The system policy is set by the policy module, and consists of a set of specific policy choices (e.g., abort processes whose value functions have fallen below zero) as well as two parameters for each process in the system. The values are K_a and K_b, and are used to make a linear transformation to the application-defined value function: $V'(t = K_a + K_bV(t))$ It is $V'(t)$ which is actually used to perform the value function scheduling computation.

3.9.3 Short-term vs. Long-term Scheduling

Using the algorithm described above in subsection 3.9.1.3, these decisions will result in a long-term high value as long as an overload condition is not maintained for a significant period of time. A critical part of the scheduling algorithm will be the computation of the probability that an overload condition, other than a transient overload, has occurred or is about to occur, which will result in making a decision about long-term reconfiguration across the system node boundaries. In addition to decisions about reconfiguration, decisions with respect to process abortion, preemption, and process scheduling in support of client processes outside the node will be made by the scheduling algorithm.

ArchOS will use a three phased approach to handling the inter-node scheduling control decisions, including reconfiguration decisions as well as decisions about scheduling processes invoked across node boundaries in support of common tasks, and including such decisions as when a process should be moved and the decision of the best destination to which it should be moved.

1. Purely local information will be utilized. This means that the attributes of the functions resident on the node will be used, but no inter-node coordination of process scheduling will take place. Decisions by processes in one arobject to request services in another arobject (either local or remote) will be made unilaterally by the requesting node.

2. Primarily local information will be utilized, augmented with value function and critical time information from the clients of the processes being scheduled. In this way, scheduling by one node on behalf of another node will be supportive of the needs of the client node as

far as possible, but no coordination of, for example, two server nodes in support of a third client node will be performed. Thus, it is possible (but hopefully unlikely) that in a heavily loaded environment two such server nodes could abort processes on behalf of clients in such a way that no client gets adequately served, even though work is being performed on its behalf.

3 Local information will be augmented by some form of negotiation among scheduling routines to ensure that work on behalf of remote client processes is coordinated, and, if abortion is necessary, a best-effort is made to do it in a way which maximizes the total system value.

3.10 Time-Driven Virtual Memory Subsystem

The Time-Driven Virtual Memory (TDVM) Subsystem is responsible for managing the primary memory page frames. It allocates those page frames among the most critical processes, as dynamically determined by the Time-Driven Scheduler (TDS) Subsystem. The basic goal of the TDS and TDVM Subsystems is to ensure that the required CPU time and primary memory space resources are both available when needed, in order to complete critical tasks within their deadlines. Furthermore, when it becomes necessary to miss some deadlines due to overload situations, TDS and TDVM attempt to allocate the oversubscribed resources to the most important processes, so as to maximize the total value of the tasks completed (see Section 3.9).

The basic goal of the TDVM Subsystem differs substantially from that of a general purpose time-sharing system's virtual memory manager. Although there are many theoretical results and a great deal of practical experience in designing virtual memory systems for general purpose computing environments, it is unclear how much of this work can be applied in the real-time (or time-driven) environment of ArchOS. Indeed, very few real-time systems to date have supported virtual memory facilities, since it is very difficult to provide real-time guarantees in the face of unexpected paging activity. Thus, the design of the ArchOS TDVM Subsystem, as outlined in this Section, can only be regarded as a preliminary version. It provides the required functionality, but many of the policies and design decisions are based on little or no supporting data or theory. This area of the ArchOS design will be one of the important focus points for further study, experimentation, and development.

3.10.1 Memory Management Policies and Techniques

In order to manage the primary memory (space) resources in a time-driven fashion, consistent with the management of the CPU (time) resources, the TDVM Subsystem uses the "working set model" as the basis for its management policies and techniques. The working set model for memory management was first described by Denning [Denning 68], and it has been the focus of a great deal of subsequent research [Denning 80]. The working set of a process is defined as the set of its virtual

memory pages which have been accessed within the last *W* units of CPU (virtual) time. Note that working sets are defined on a per-process basis, and are determined with respect to the per-process virtual times. The goal of a working set memory manager is to ensure that the complete working set for each active process is resident in primary memory. If there are too many active processes, some will have to be swapped out to secondary memory (and thus become inactive), to ensure that all of the remaining working sets will fit within primary memory. This guarantees that the active processes will execute efficiently, with a minimum number of page faults (i.e. "thrashing" is avoided).

Since a working set memory manager determines which pages are to be resident in primary memory on a per-process basis, it is called a "local" memory management technique. This is in contrast to "global" techniques, which are only concerned with the *real* time since a page was last accessed, regardless of which process it belongs to. For the purposes of time-driven virtual memory management, a local technique is expected to be much more effective in allocating the primary memory resources to the most important processes. This is because it can account for variations in the urgency of processes, based on their deadlines and the penalties for missing them, and ensure that the most urgent processes have their working sets completely resident.

A key factor affecting the operation of a working set memory manager is the choice of *W*, the working set parameter or working set window size. In traditional time-sharing environments, it has been found that using a single value of *W*, the same for all processes, works almost as effectively as selecting the window size on a per-process basis. Also, the overall paging behavior of the system is not very sensitive to small changes in *W*, although *W* must be tuned somewhat to yield performance that is close to optimum.[56] The situation in a time-driven environment is somewhat different, since the goal is to ensure that deadlines, especially all of the more important deadlines, are met, and there is much less concern with overall system throughput. In this case it could be beneficial to vary the working set window sizes, depending on the criticality of the individual processes. Furthermore, since the criticality of a process can vary with time, it may be useful to dynamically vary its window size as well.

Since the value of *W* depends upon the criticality of the individual process (how near it is to its deadline, and the penalty for missing it), only the TDS Subsystem is in a position to specify the varying window sizes. Only a few, discrete window sizes, corresponding to multiples of some standard

---

[56] If *W* is too small excessive paging will occur, because it will not capture the "natural" working sets for many processes. On the other hand, if *W* is too large, excessive swapping will occur. This is because the working sets will be larger, due to pages taking longer to pass out of the working set window, and fewer working sets can then be fit in primary memory. Getting *W* close to optimum involves balancing these two effects so as to maximize overall system throughput.

window size, w, should actually be needed. An adequate set of sizes might be $\{w, 2w, 4w, \ldots, 128w\}$. Note that w is a global system tuning parameter, corresponding to the single window size of a standard, time-sharing, working set memory manager. TDS informs TDVM of the various working set window sizes at the same time it provides TDVM with the ordered list of runnable processes, i.e. as a result of calling the *Schedule* primitive (see Section 3.2.2.4).

For the case of time-sharing working set memory managers, it has been found that maintaining just the working set sizes (number of pages) across process swaps, without recording the individual pages of the working sets, still yields adequate performance. The working set pages are simply faulted back in on demand, after the process has been swapped back in. However, in a time-driven system such paging behavior may not be acceptable, especially if the process is nearing its deadline. As a result, TDVM will maintain the actual lists of working set pages for all processes, whether active or swapped. This will allow the working set of a process to be efficiently reloaded at swap-in time.

The choice of which working sets should be resident in primary memory, and which should be swapped-out, will be made by TDVM based on the order of the processes in the scheduling list, which is returned by the TDS *Schedule* (or *RequestSwapList*) primitive. In general, the most urgent processes will be at the head of the scheduling list, and TDVM will attempt to load and maintain the working sets of as many of those processes as possible. Since it is expected that most processes will gradually rise through the scheduling list as their deadlines approach, this technique should result in the preloading of most working sets, prior to the time the associated processes have to run. If a working set has to be swapped-out in order to make space for one that is to be swapped-in, either the unswapped process closest to the end of the scheduling list, or the process that blocked least recently, will be selected for swapping-out.[57] If the unswapped process that blocked least recently has been blocked for more than some threshold amount of time, it will be the one selected for swapping-out.

For the processes which are currently active, techniques are required for determining which pages belong to their working sets. At the time that a new process is created, there is no execution history available to indicate what its initial working set ought to be. In order to reduce the number of initial page faults, and hence improve the efficiency of process "loading" and startup, TDVM will allow an initial working set to be specified. This initial working set could either be determined heuristically (for example, x percent of the User Text and y percent of the User Data), or it could be based on information obtained through previous executions of the process.

---

[57] The scheduling list will contain all processes which are "sleeping" (due to *Delay* or *Alarm* primitives), but any processes blocked for other reasons (such as *Accept* primitives) will not appear in that list.

As a process runs, TDVM must dynamically adjust the process' working set, to track the process through different phases of its execution. The primary way that pages get added to a working set is through page faults, i.e. as new pages are accessed they get paged-in "on demand". If TDVM detects sequential paging activity in one of the data segments of a process' address space, it can attempt to do sequential pre-paging. This technique can be very effective in improving the performance of sequentially accessed File Arobjects, and in many other situations as well. One other way that pages can get added to a working set is if the working set window size is increased.[58] When TDVM detects an increase in the window size (by checking the value in the scheduling list, obtained through the TDS *Schedule* primitive), it must scan through the "last accessed times" of all pages in the address space. This will allow it to determine which pages should be added to the working set (paged-in), in addition to those already there.

The removal of pages from a working set occurs when pages fall outside of the working set window. This can happen either as a result of the process' virtual time advancing more than *W* units since one or more of the working set pages were last accessed, or as a result of the working set window size decreasing. A decrease in the window size is detected in the same manner as an increase, but in this case only the current working set pages need be scanned to determine which, if any, are to be removed from the working set. Determining the last accessed times for pages of the working set can only be done approximately, assuming no specialized virtual memory hardware other than USED flags is available. It is accomplished by scanning the USED flags of the working set pages every time *w* units of virtual time have passed.[59] For every page in which the USED flag is set, that flag is cleared and the last accessed time is updated to the current virtual time. For any page in which the USED flag was clear, the last accessed time is checked to see if it falls outside of the working set window. If so, that page is removed from the working set.[60]

One other major responsibility of the TDVM Subsystem is management of the free page frame pool. A free page is any primary memory page which does not belong to one of the currently active working sets. Such a page can be in any one of three different states:

1. On the Page-Out List: It corresponds to an existing page in a secondary memory page set, but it has been modified and hence must be written back to the page set before it can be reused.

---

[58] This happens when TDS determines that it has now become more important to avoid page faults, because the process is nearing its deadline and the penalty for missing the deadline is high.

[59] Recall that *W*, the working set window size, is some multiple of *w*, the base window size.

[60] Note that, in practice, working set scans could be initiated at the time of processor rescheduling decisions (calls to *Schedule*), assuming that such decisions occur at intervals of at most *w* units of real time. In that case the current process' working set need only be scanned if the process has accumulated at least *w* units of virtual time since it was last scanned.

2. On the Reclaim List: It corresponds exactly to an existing page in a secondary memory page set, and hence it is free to be reused without first writing it back to the page set.

3. On the Free List: It is completely free to be reused, since it does not correspond to any existing page in a secondary memory page set.

When a page fault occurs, TDVM checks the Page-Out and Reclaim Lists to see if the required page is already in primary memory, and hence can be quickly "reclaimed". If so, the time and expense of reading the page from its secondary memory page set can be avoided. Otherwise a free page must be selected to hold the contents of the page as it is read from the page set. Pages are selected first from the Free List, but if it is empty, then they will be selected from the Reclaim List. The Reclaim List is maintained in FIFO order, so that those pages which have been free for the longest time will be selected first for reuse. If the Reclaim List is also empty, then a page from the Page-Out List will have to be selected. Of course, that page will have to first be written back to its corresponding page set, before it can be reused. The Page-Out List, like the Reclaim List, is maintained in FIFO order, so the page that has been free the longest will be the first to be reused. In the event that no free page can be found, i.e. the Page-Out List is also empty, then TDVM must initiate the swapping-out of a process in order to make some free pages available.

To help avoid the situation of having to page-out or swap-out at page fault time (i.e. to help reduce the elapsed time for handling a page fault), TDVM attempts to maintain a mimimum number of pages in the Free List and Reclaim List combined. When the number of available pages in these lists drops below a threshold (approximately five percent of the primary memory page frames), TDVM will initiate page-out operations in order to move some pages from the Page-Out List to the Reclaim List. If there are still not enough free pages available, then TDVM will initiate a swap-out operation in order to free all of the working set pages from some process (the least urgent one). Note that checking the free page threshold and initiating page-out and swap-out operations can be done at the time of processor rescheduling decisions, just as for the case of initiating working set scan operations.

3.10.2 Time-Driven Virtual Memory Subsystem Primitives

The TDVM Subsystem provides the following set of primitives, primarily for use by the Arobject/Process Management (A/PM) Subsystem:

```
val = VMPrePage(asid, vpa, nvp)
val = VMLock(asid, vpa, nvp)
val = VMUnlock(asid, vpa, nvp)

pid = VMPageFault(asid, vpa)
pid = VMSchedule()

val = VMFlush(asid, vpa, nvp)
val = VMSwap()
gpsid = VMFreeze(asid)
val = VMUnfreeze(gpsid [, asid])

val = VMMove(asid, vpa, desired-vpa)
val = VMZero(asid, vpa, nvp)

val = VMDestroy(asid)
val = VMFree(asid, vpa)

val = VMStatus(statusbuffer [, asid])
val = VMRestart()
```

BOOLEAN val — TRUE if the specified operation is completed successfully; otherwise FALSE.

PID pid — The process ID of the process which is to be run at this time.

GPSID gpsid — Global Page Set ID, which identifies the *temporary* page set containing the frozen (swapped) address space descriptor information. It includes the ID of the logical disk containing the page set, the page set type (TEMPORARY), and the unique ID of this page set within the logical disk.

ASID asid — Address Space ID, which identifies the address space to be operated upon. The corresponding process ID can be easily obtained from an ASID, and vice versa.

VIRT-PAGE-ADDRESS vpa, desired-vpa
A virtual page address within an address space.

INT nvp — The number of virtual pages involved in the operation.

VMSTATUSBUFFER *statusbuffer
The buffer address for returning status information.

On Error: Error conditions are indicated by the use of special return values. The details concerning the precise nature of an error condition are provided in the Kernel Error Block.

The *VMPrePage* primitive initiates the paging-in of the specified set of virtual pages (*nvp* pages, beginning with page *vpa*), in address space *asid*. This is useful for setting up an initial working set of pages for a newly created process, since otherwise there is no execution history to indicate what the working set ought to be. The *VMLock* primitive is similar to *VMPrePage* in that it causes the specified set of virtual pages to be paged-in, if not already in primary memory. However, *VMLock* is synchronous in the sense that it will not return until all of the specified pages are actually present in primary memory. Furthermore, those pages will be "locked" in primary memory, i.e. they will not be considered eligible for removal from the address space's working set, regardless of how long it has been since they were last accessed. *VMUnlock* will "unlock" the specified set of virtual pages, making them eligible for removal from the working set, and allowing the corresponding physical page frames to be subsequently reused.

VMPageFault is the means by which the TDVM Subsystem is notified of page faults. It is assumed that the specified virtual page (*vpa*) is indeed one of the allocated pages of address space *asid*, and hence the page fault is truly due to accessing a valid page that isn't resident in primary memory (rather than a protection or invalid address fault). *VMPageFault* first checks to see if the missing page is already present somewhere in primary memory. This would be the case if the page had been in primary memory earlier, was "freed" due to inactivity, but the associated page frame had not yet been reused. It would also be the case if the page is shared and is already resident as part of another process' working set. If the missing page is found in primary memory, the page fault is handled very quickly, with no need to switch processes. In this case *VMPageFault* will return the ID (*pid*) of the current process, which is the one that encountered the page fault. Otherwise *VMPageFault* must initiate a page-in operation. Since reading a page from a secondary memory page set can take tens of milliseconds (or even hundreds of milliseconds if paging-out is required before space is available for the new page), a process switch is usually desireable when paging-in is required. In such cases, *VMPageFault* will return the ID of the process which should now run in place of the faulting process.

VMSchedule is intended to be the primary means by which the Arobject/Process Manager determines which process is to be running at the present time. *VMSchedule* calls the *Schedule* function of the Time-Driven Scheduler, to obtain the ordered list of processes which are to be run next. From this list the TDVM Subsystem can determine which address space working sets will be required in primary memory in the near future, allowing it to initiate any necessary page-in operations. Related to this, *VMSchedule* is also responsible for scanning the working set pages of the current process, to see if any of them have not been accessed recently and can thus be removed from the working set. This scanning operation is only initiated if the process has accumulated sufficient virtual (CPU) time since its working set was last scanned. *VMSchedule* returns the ID of the process which is

to run now. It guarantees that the address space for that process is the active one (see the description of *ASActivate* in Section 3.2.2.5).

The *VMFlush* primitive ensures that the specified set of virtual pages from the given address space are all "clean", i.e. any primary memory images of these pages are paged-out, if they have been flagged as MODIFIED. *VMFlush*, like *VMLock*, is synchronous. It will not return until all of the modified pages have been written out to their corresponding page sets. *VMFlush* is primarily intended to support the *FlushPermanent* primitive of the Arobject/Process Manager (see Section 3.3.3). It is also used when "deactivating" arobjects, to ensure that all modifications have been flushed out to the appropriate page sets before the address spaces are destroyed. Unless directed to flush modified pages through *VMFlush*, the TDVM Subsystem will only initiate page-out activity when primary memory begins to fill up, and some of the modified but not recently accessed page frames have to be reused.

VMSwap provides a mechanism through which the Arobject/Process Manager can explicitly request the TDVM Subsystem to "swap out" one of the existing but currently idle address spaces. Ordinarily, the TDVM Subsystem will only swap out an address space if the combined working sets of all of the existing address spaces overflow primary memory. In that case TDVM calls the Time-Driven Scheduler's *RequestSwapList* function, to obtain the ordered list of swappable processes. TDVM selects the most appropriate candidate from this list.[61] All of the pages from the selected process' working set are released, so that they can be paged-out and reused as required. TDVM then writes all of the relevant address space descriptor information (including the list of working set pages) into a *temporary* page set.[62] The address space is then destroyed (using *ASDestroy*), which frees the associated address space resources for use by others. This freeing of address space resources is the primary reason for making *VMSwap* available to the Arobject/Process Manager. It provides a mechanism for recovering from "space exhausted" problems when creating or growing address spaces.

VMFreeze is quite similar to *VMSwap*, except that it allows a specific address space (*asid*) to be selected for "swapping". If that address space is already swapped out, the global page set ID (*gpsid*)

---

[61] The least "urgent" process, for which the expected continuation time is farthest in the future, and which hasn't already been swapped out, is selected. However, processes containing locked address space pages are not considered eligible for swapping.

[62] It is not necessary to record all of the address space information. Only the Private Region segment descriptors, and the EXISTS/COPIED flag for each page in those segments need be saved. If this was the last process from the associated arobject which was still resident on this node and not yet swapped out, then the Shared Region segment descriptors and the corresponding EXISTS flags will also have to be saved.

of the *temporary* page set containing the address space descriptor is returned, after first ensuring that all MODIFIED pages from that address space have been flushed.[63] If the specified address space is not currently swapped out, it is flushed and swapped (even if it contains locked pages), and the *gpsid* of the *temporary* page set containing the address space descriptor is returned. *VMFreeze* provides support for the Arobject/Process Management Subsystem's *FreezeArobject* and *FreezeProcess* primitives (see Section APMSEC). It is also used when migrating processes from one node to another.

Following a *VMFreeze* operation, the TDVM Subsystem removes all record of address space *asid*, except for the page sets on secondary memory. The *VMUnfreeze* primitive can be used to restore a previously frozen address space (indicated by *gpsid*, the ID of the page set containing the address space descriptor), to the swapped state (see Figure 3-33). *VMUnfreeze* provides support for the *UnfreezeArobject* and *UnfreezeProcess* primitives of the A/PM Subsystem (see Section APMSEC). Optionally, an address space can be given a new ID (*asid*) at the time it is unfrozen. This is useful when migrating processes, since it allows the address space to be associated with a new process ID on the new node.

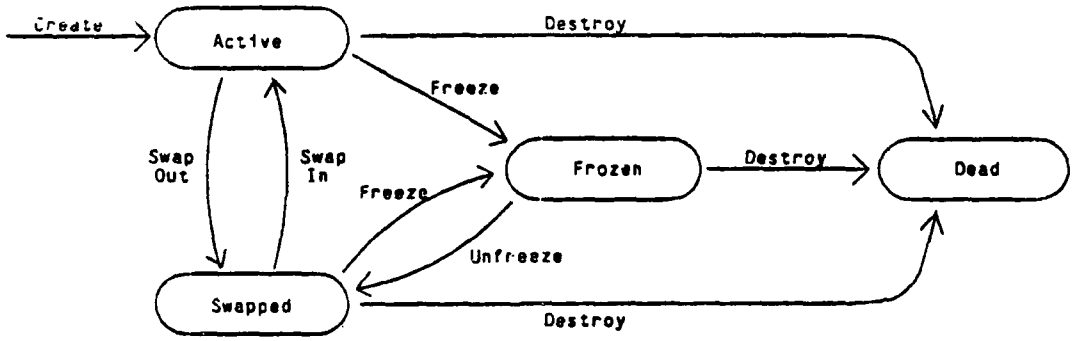


Figure 3-33: Life Cycle of an Address Space

VMMove is the Arobject/Process Manager's interface to the *ASMove* function, which changes the location of an existing segment in address space *asid* from *vpa* to *desired-vpa*. This function must go through the TDVM Subsystem, so that TDVM can update any references it may have to the affected virtual pages. See Section 3.2.2.5 for more details concerning the *ASMove* primitive. *VMZero*

[63] When an address space is swapped, all of its MODIFIED pages are flagged for paging-out, but may not be flushed immediately. A frozen address space, on the other hand, is guaranteed to have been completely flushed.

provides an efficient mechanism for "zeroing" complete pages of an address space. It does this by clearing the EXISTS flags for the specified virtual pages, so that the next time any of these pages are accessed they will first be zero-filled. If the pages to be zeroed already exist on the corresponding page set (the EXISTS flags were set), then *VMZero* will also call the appropriate *PSZero* or *APSZero* primitive to zero (or free) the pages from the page set. One of the main purposes of the *VMZero* primitive is to provide efficient support for the *ZeroFile* primitive, which is provided by the File System Client Interface (see Section 3.6).

VMDestroy and *VMFree*, like *VMMove*, are the A/PM Subsystem's interface to the corresponding Address Space Management Primitives (*ASDestroy* and *ASFree*). *VMDestroy* completely destroys the specified address space (*asid*), while *VMFree* deletes a segment (indicated by *vpa*) within address space *asid*. These functions must go through the TDVM Subsystem, so that TDVM can remove any references it may have to the affected address space or segment, and free any corresponding primary memory page frames. See Section 3.2.2.5 for more details concerning the *ASDestroy* and *ASFree* primitives.

The *VMStatus* primitive is used to obtain general information concerning the current status of the TDVM Subsystem. This includes the number of active and swapped address spaces, the average size of the address space working sets, the number of free pages available, and the total number of page-in, page-out, and page-reclaim (avoided or fast page-in) operations. This information can be used for monitoring the system's virtual memory activity, so that the TDVM Subsystem can be tuned to provide better performance. If an optional address space ID (*asid*) is specified, then *VMStatus* will return virtual memory information related to that particular address space. This includes its status (SWAPPED or ACTIVE), working set size, and page-in, page-out, and page-reclaim statistics.

The *VMRestart* primitive is the initialization operation for the TDVM Subsystem (on a particular node). It is assumed to be called automatically as the first TDVM operation, upon restarting (rebooting) a failed node. Its sole purpose is to create the TDVM worker (kernel) processes, and initialize the virtual memory management data structures. These TDVM internal processes and data structures are described in the following section.

3.10.3 Components of the Time-Driven Virtual Memory Subsystem

The TDVM Subsystem is implemented as a kernel arobject, with a separate instance on each node of the distributed computer system. Each instance is solely responsible for the management of the primary memory page frames on its own node, and has no need to communicate or cooperate with any of its peers on other nodes. Each instance of the TDVM Subsystem consists of three processes

and five main data structures, as illustrated in Figure 3-34. The TDVM Manager process provides almost all of the subsystem's functionality, and is the only process which can access and modify the main data structures. The sole purpose of the Page-In Worker and Page-Out Worker processes is to allow paging (page set read and write) operations to be performed asynchronously with other TDVM Subsystem processing. The two worker processes are basically surrogates for the TDVM Manager. They wait for page set operations to be completed on behalf of the TDVM Manager, allowing the Manager to continue handling other virtual memory related operations.

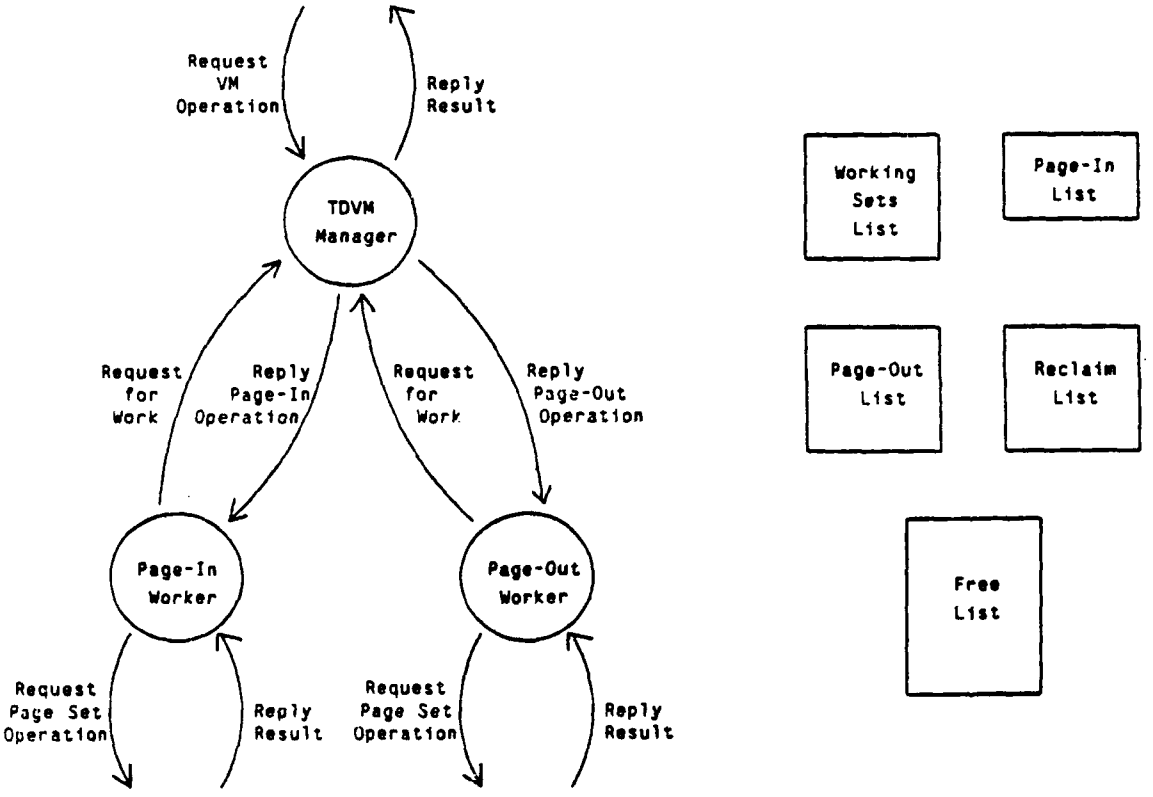


Figure 3-34: Components of the Time Driven Virtual Memory Subsystem

The first main data structure indicated in Figure 3-34 is the Address Space Working Sets List. This data structure is illustrated in more detail in Figure 3-35. The Address Space List contains an entry for every address space that is known to the TDVM Subsystem (whether it is currently active or swapped out). Each entry indicates the current *state* of the address space (ACTIVE or SWAPPED). If SWAPPED, the global page set ID (*gpsid*) of the *temporary* page set containing the address space's

descriptor is provided. See the second entry in the Address Space List in Figure 3-35 for an example. Otherwise the entry contains information concerning the address space's current working set of virtual pages.

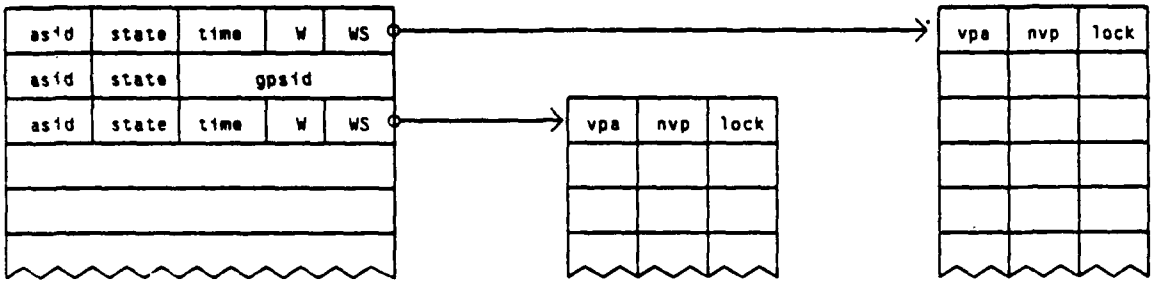


Figure 3-35: Address Space Working Sets List

For ACTIVE address spaces, the "last scan time" (*time*) and working set window size (W) are provided, in addition to the actual list of virtual pages which comprise the working set. The last scan time is the virtual (CPU) time at which the address space was last scanned to determine which pages belong to the working set (were accessed recently). The *W* parameter defines "recently", by specifying the virtual time frame during which a page must have been accessed, in order to be considered a member of the current working set. The list of working set pages is arranged in clusters, where each cluster is indicated by a starting address (*vpa*) and its size (*nvp*). Due to the "locality of references", clustering of working set pages is expected to be quite common. Thus, this representation of working sets will save space, and it will also improve the efficiency of page-in, page-out, swap-in, and swap-out operations, since entire clusters can be read and written at a time. Also associated with each cluster is a *lock* flag, which indicates whether or not those pages are locked in primary memory (as a result of a previous *VMLock* primitive).

The second main data structure of the TDVM Subsystem is the Page-In List. This list is actually maintained as two separate lists, one for urgent page-in operations (those resulting from page faults), and another for the less urgent "pre-paging" operations (including the swapping in and expanding of the working sets of address spaces which have increased in "urgency"). The use of two lists allows the urgent page-in operations to be given precedence over pre-page operations. The two Page-In Lists have identical structures, as illustrated in Figure 3-36.

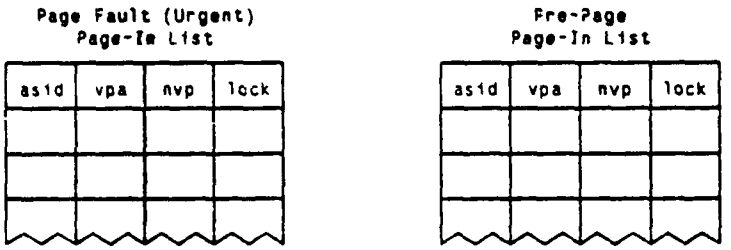


Figure 3-36: Page-In Lists

Each entry in a Page-In List indicates the address space (*asid*) and the virtual page (*vpa*) of a virtual page that is to be paged-in. The corresponding page set and page number can then be determined using the *ASGetGPSID* or *ASGetSTE* primitives (see Section 3.2.2.5). Each entry can also indicate a range or cluster of *nvp* virtual pages, beginning with virtual page *vpa*. Normally, *nvp = 1* for entries in the Urgent List, since this will allow the corresponding process to continue executing as quickly as possible.[66] However, the clustering of page-in operations in the Pre-Page List will help improve the overall paging throughput. Each entry in a Page-In List also contains a *lock* flag, which indicates whether that cluster of pages is to be locked into the corresponding address space's working set, after it has been paged-in.

Two of the other main data structures of the TDVM Subsystem are the Page-Out List and the Reclaim List. These two structures are closely related in that they both contain information about primary memory page frames which no longer belong to any address space's working set. The only difference between the two lists is that the pages in the Page-Out List have been modified, and hence they must be written back to their corresponding page sets before they can be reused. The Page-Out List and the Reclaim List each have the same logical structure, as illustrated in Figure 3-37.

Each entry in the Page-Out List or Reclaim List represents a cluster of *npages* physical page frames, beginning with *ppa*. This cluster corresponds to the *npages* of page set *gpsid*, beginning with page *pnum*.[67] The Page-Out List and Reclaim List together allow the TDVM Subsystem to quickly "reclaim" pages which have not yet been reused, thus avoiding unnecessary page-in operations. Although

[66] Some experimentation is needed to determine in which circumstances, if any, it might be desireable to perform clustered urgent page-in operations.

[67] Once again the clustering of pages, especially in the Page-Out List, should help improve paging throughput.

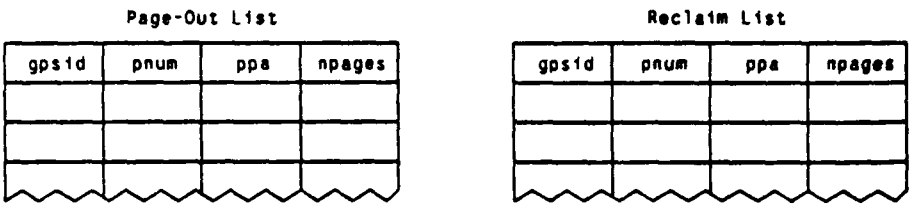


Figure 3-37: Page-Out and Reclaim Lists

conceptually they are simple lists (as illustrated in Figure 3-37), in practice the Page-Out List and Reclaim List would be sorted into binary search trees, using (*gpsid*, *pnum*) as the key. This would make the searching for reclaimable pages, as well as the insertion of new pages into existing clusters very fast. In addition, each entry in the Page-Out List and Reclaim List would be threaded in FIFO order on its respective "reuse queue". This ordering would allow the least recently used pages to be selected for reuse, whenever the pool of "free" page frames was exhausted.

The final main data structure of the TDVM Subsystem is the Free Page Frame List, which is illustrated in Figure 3-38. Each entry in the Free List has a very simple structure, and represents a cluster of *npages* physical page frames, beginning with *ppa*. Each cluster is completely free to reuse, since it does not correspond to any existing virtual address space pages. In practice, the Free List would be sorted into a binary search tree, using *ppa* as the key. This would make the insertion of newly freed page frames into existing clusters very fast. In addition, each entry would be threaded on an appropriate list according to its cluster size (*npages*). There are 32 separate cluster size lists, one for each power of two. This arrangement helps reduce primary memory "fragmentation", allowing page-in clusters of the appropriate size to be found very quickly.

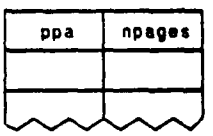


Figure 3-38: Free Page Frame List

3.10.4 Interaction With Arobject/Process Manager and Time-Driven Scheduler

The TDVM Subsystem interacts extensively with both the Arobject/Process Manager and the Time Driven Scheduler, on the local node. Figure 3-39 illustrates the primary "uses" relationships among these subsystems. A/PM interacts with TDVM in two main ways, either directly through the invocation of TDVM primitives, or indirectly through the manipulation of address spaces. Since most address space manipulations are actually performed by TDVM, and TDVM must maintain consistency between its own data structures and those of the Address Space Management Routines, A/PM is quite restricted in terms of the address space operations it is allowed to perform. In particular, certain operations (*ASDestroy*, *ASFree*, *ASMove*) must be invoked by A/PM via the corresponding TDVM primitives, as noted earlier. However, address space creation (*ASCreate*) and growth (*ASAllocate*, *ASExpand*) can both be handled directly by A/PM, since TDVM will learn about them automatically as the new address space and pages are used.

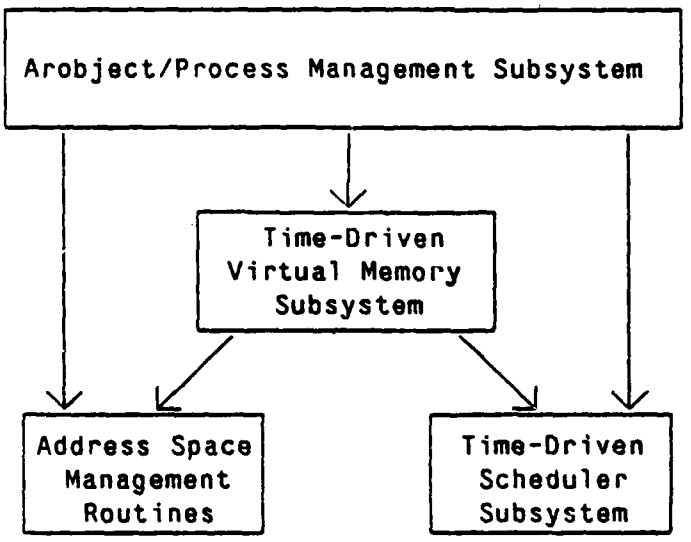


Figure 3-39: TDVM, A/PM, and TDS Subsystem Interactions

Besides address space manipulations, the other major area of A/PM and TDVM interaction is process scheduling. The determination of which process is the best choice to be running on a node at any given time is ultimately the responsibility of the Time-Driven Scheduler. However, TDS is not aware of the status of the process address spaces (ACTIVE or SWAPPED), and thus it could select a

process which TDVM has not yet had time to swap back in.[68] In order to coordinate properly between TDVM and TDS on the choice of which process to run at the present time, A/PM must always use the *VMSchedule* primitive, rather than directly invoke the TDS *Schedule* primitive. Figure 3-40 illustrates the normal sequence of events which occur when A/PM must reschedule the processor, due to the current process blocking or completing.

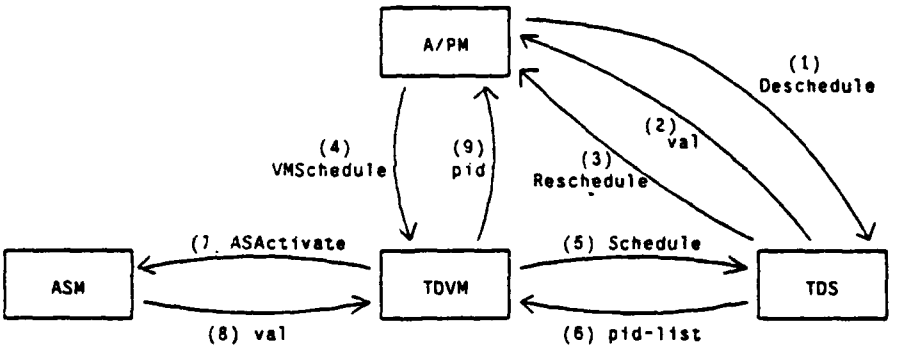


Figure 3-40: Normal Processor Rescheduling Sequence

A/PM first invokes the TDS *Deschedule* primitive, and after it completes (returns *val*) A/PM enters the "idle state", waiting for a "*Reschedule*" signal to indicate that another process is ready to run. When TDS has determined which process to run next, it sends the *Reschedule* signal (interrupt) to A/PM.[69] Upon receipt of the *Reschedule* signal, AP/M calls *VMSchedule* to determine exactly which process is to be run next, and to switch to its associated address space. *VMSchedule* itself calls the TDS *Schedule* primitive, to see which process the scheduler has selected. Assuming the selected process is not swapped out, TDVM activates the process' address space (by calling *ASActivate*), and returns the ID of the process (*pid*) to A/PM. A/PM then completes the switch to process *pid*, and sets it running.

If the process selected by the TDS *Schedule* primitive (the first process returned in *pid-list*) is currently swapped out, *VMSchedule* will initiate the swapping-in of that process, and select a different

[68] This should seldom happen, but it can occur if all of the most urgent processes block in rapid succession, waiting for various events. The best choice among the remaining "ready" processes could then be one that is still swapped out, since TDVM had not anticipated the need to run it so soon. Another way this can happen is if a process which was blocked for a long time and got swapped out suddenly becomes ready to run again, and it is immediately selected as the most urgent process.

[69] If another process is already available and ready to run, there should be essentially zero delay before TDS sends the *Reschedule* signal.

pid from *pid-list*. To be selected, a process must be the first one in *pid-list* which is not currently swapped out, awaiting completion of a page-in operation, or put on "*hold*" by TDS.[70] If no appropriate process can be found, *VMSchedule* returns BAD-PID to A/PM. This informs A/PM that it should return to the idle state, rather than switch processes at this time.

In addition to rescheduling the processor when the current process blocks or completes, preemptive rescheduling can also occur. Preemptive rescheduling due to an urgent process unblocking works almost identically to the normal rescheduling sequence outlined above, and illustrated in Figure 3-40. The only difference is that A/PM invokes the TDS *SetScheduleInfo* primitive (rather than *Deschedule*), upon the occurrence of an event for which a process is waiting (blocked). This informs TDS that the previously blocked process is now schedulable, so that TDS can determine when it should be run. In the meantime, A/PM can allow the process which was running at the time of the event to continue executing. If TDS determines that the newly unblocked process should be run immediately, it will send a *Reschedule* signal to A/PM. A/PM will then save the state of the current process, and initiate the standard *VMSchedule* sequence discussed above. Similarly, if at any point TDS wishes to preemptively reschedule the processor (because an urgent process has just completed its delay, or there has been some other significant change in the value functions of processes), it can do so by simply sending a *Reschedule* signal to A/PM.

Two other reasons for rescheduling the processor, of particular relevance to TDVM, are the occurrence of page faults and the completion of page-in and swap-in operations. The sequence of events which occurs upon page fault is illustrated in Figure 3-41. A process execution fault first comes to the attention of A/PM. If it is a page fault, A/PM saves the state of the current process, and invokes the *VMPageFault* primitive of TDVM. If a fast page reclaim is possible, TDVM handles it immediately and returns the *pid* of the current process, so that A/PM can continue its execution. Otherwise, TDVM initiates a page-in operation, and calls *Schedule* to determine which process should be run while waiting for the page-in to complete. Note that since the faulting process has not been *Descheduled*, it will still appear in the *pid-list* returned by *Schedule*, but it will be ineligible for selection because it is awaiting completion of the page-in operation. If no appropriate process is available, BAD-PID is returned to A/PM, indicating that the processor should remain idle at this time. Otherwise, the address space of the selected process is activated, using *ASActivate*, and its *pid* is returned to A/PM in order to set it running.

---

[70] The *hold* flags are returned along with the *pids* in *pid-list*. They indicate which processes are not to be run at present, either because they are still "sleeping", or because there is a penalty for running them too soon.

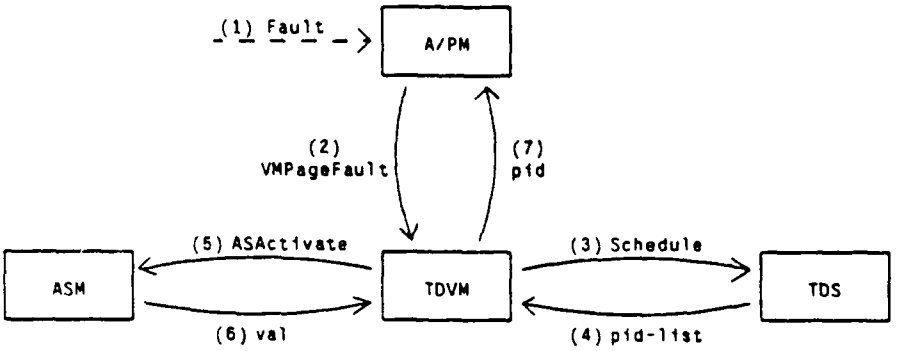


Figure 3-41: Page Fault Sequence

The sequence of events which occurs upon the completion of a page-in or swap-in operation is illustrated in Figure 3-42. TDVM is notified of such an event through a new "Request for Work" from its Page-In Worker process (see Figure 3-34). In response to this, TDVM initiates the standard (preemptive) rescheduling sequence (as discussed above), by sending a *Reschedule* signal to A/PM. Note that the process for which the page-in or swap-in operation has just been completed should still appear in the *pid-list* returned by *Schedule*, but now it will be eligible for selection as the process to be run next.

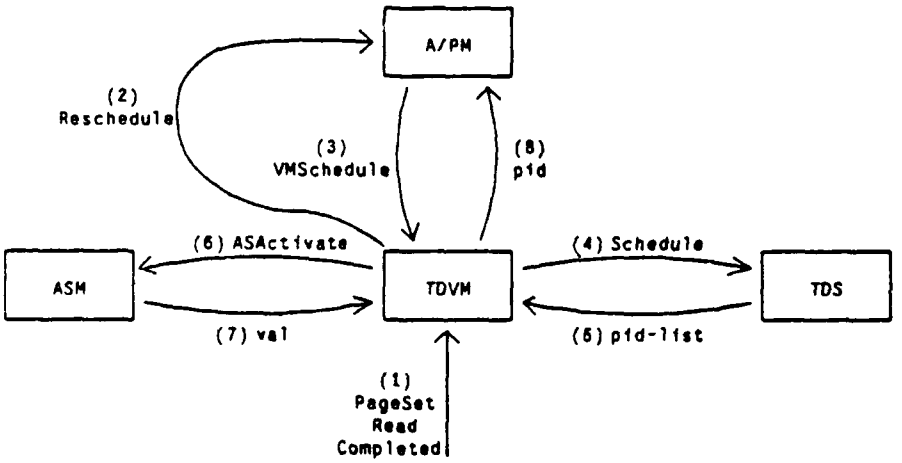


Figure 3-42: Page-In or Swap-In Completion Sequence

3.11 System Monitoring and Debugging Subsystem

The system monitoring and debugging subsystem provides various abilities to monitor and control behavior of cooperating arobjects and processes during the execution time. The monitoring and debugging manager exists on each node and has a special privilege to *freeze* or *unfreeze* an activity of the arobject or process.

3.11.1 Monitoring and Debugging Management

The actual operations for the monitoring and debugging operations are performed in the arobject/process or communication subsystems. For instance, *Freeze*, *UnFreeze*, *Fetch*, *Store* operations on an arobject/process is performed at the arobject/process manager. However, *Freeze* and *UnFreeze* operations for a specific node or for all applications are initiated at the monitoring and debugging manager. Similarly, monitoring on the communication activity is supported by the communication subsystem.

The following ArchOS primitives are supported for the monitoring and debugging management for a client.

---

val = **FreezeAllApplications()**
val = **UnfreezeAllAplicatons()**
val = **FreezeNode(node-id)**
val = **UnfreezeNode(node-id)**

BOOLEAN val TRUE if the primitive was executed properly; otherwise FALSE.

NODE-ID node-id The node id indicates the actual node which will be stopped.

A *FreezeAllApplications* primitive stops the entire activities of caller's application, and a *FreezeNode* primitive halts all of the client's activities in a specific node. To resume client's application, *UnfreezeAllApplications* or *UnfreezeNode* will be used.

3.11.2 Monitoring/Debugging Protocol

When a system-wide Monitoring/Debugging request such as *FreezeAllApplications* is issued, a Monitoring/Debugging Manager's worker becomes a coordinator among the other Monitoring/Debugging Managers and propagates the request to the others.

References

[Bernstein 83] P. A. Bernstein, Goodman, N.; Hadzilacos, V.
Recovery Algorithms for Database Systems.
Technical Report TR-10-83, Harvard University, 1983.

[Comer 79] Comer, D.
The Ubiquitous B-tree.
ACM Computing Surveys 11(2):121-137, June, 1979.

[Conway 67] Conway, Maxwell, and Miller.
Theory of Scheduling.
Addison-Wesley, 1967.

[Denning 68] Denning, P.J.
The Working Set Model for Program Behavior.
Communications of the ACM 11(5):323-333, May, 1968.

[Denning 80] Denning, P.J.
Working Sets Past and Present.
IEEE Transactions on Software Engineering SE-6(1):64-84, January, 1980.

[Eswaran 76] Eswaran, K. P., J. N. Gray, R. A. Lorie and I. L. Traiger.
The Notions of Consistency and Predicate Locks in a Database System.
Comm. ACM 19(11), November, 1976.

[Garey 79] Garey, M. R.; Johnson, D. S.
Computers and Intractability: A Guide to the Theory of NP-Completeness.
W. H. Freeman, San Francisco, 1979.

[Gray 81] Gary, J. N.
The Trasaction Concept: virtues and limitations.
Technical Report, , 1981.

[Jensen 83a] E. Douglas Jensen.
Decentralized System Control.
may, 1983
Final Report for RADC, Contract Number F30602-81-C-0297, Department of Computer Science, Carnegie-Mellon University.

[Jensen 83b] E. Douglas Jensen, et al.
Reconfigurable C^2 DDP System.
November, 1983
Technical Proposal to RADC, PR No. B-4-3137, Department of Computer Science, Carnegie-Mellon University.

[Jensen 84] Jensen, E. D. and Pleszkoch, N.
ArchOS: A Physically Dispersed Operating System -- An Overview of its Objectives and Approach.
IEEE Distributed Processing Technical Committee Newsletter, Special Issue on Distributed Operating Systems , June, 19E ..

[Jensen 85] E. Douglas Jensen, H. Tokuda, et al.
Functional Description of ArchOS (Archons Operating System).
January, 1985
Technical Report for RADC, Contract No. F30602-84-C-0063, Department of Computer Science, Carnegie-Mellon University.

[Liu 73] Liu, C. L.; Layland, J. W.
Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.
Journal of the Association for Computing Machinery 20(1):46-61, January, 1973.

[Locke 85] C. Douglass Locke, Tokuda, H; E. Douglas Jensen.
A Time-driven Scheduling Modle for Real-Time Operating Systems.
To appear in Proceedings of Real-time Systems Symposium , 1985.

[Mitchell 82] Mitchell, J., and Dion, J.
A Comparison of Two Network-Based File Servers.
Communications of the ACM 25(4):233-245, April, 1982.

[Moss 81] Moss, J. E. B.
Nested Transactions: An Approach to Reliable Distributed Computing.
PhD thesis, Massachusetts Institute of Technology, April, 1981.

[Sha 85] Sha, L.
Modular Concurrency Control and Failure Recovery -- Consistency, Correctness and Optimality.
PhD thesis, Carnegie-Mellon University, 1985.

[Sturgis 80] Sturgis, H., Mitchell, J., and Israel, J.
Issues in the Design and Use of a Distributed File System.
Operating Systems Review 14(3):55-69, July, 1980.

[Tokuda 85] Tokuda, H.
A Compensatable Atomic Objects in Object-oriented Operating Systems.
To appear in Proceedings of Pacific Computer Communicaiton Symposium , 1985.

[Wulf 81] Wulf, W. A.; Levin, R.; Harbison, S. P.
HYDRA/C.mmp: An Experimental Computer System.
McGraw-Hill, 1981.

4, Resource Management Theory

4.1 Best Effort Decision Making Overview

Best Effort Decision Making Overview

1 Best Effort Decision Problem Statement

The purpose of any operating system is to present a virtual computer interface to a set of application programs such that the user can effectively utilize the resources of the computer to solve a particular problem. This virtual computer has, as its goal, a presentation of the resources of the underlying real computer in a form such that they can be efficiently and correctly used.

In a uniprocessor-based system, the management of resources can be performed taking into account the complete state of the system, since it is generally assumed that the state is entirely available at a single location. Algorithms for the management of resources in such systems have been extensively investigated, and the solutions are embodied in many existing systems.

In the case of a distributed system, the resources are also distributed, and the knowledge of the system state is not available at any one place. In fact, the very concept of an instantaneous state of the resources is essentially meaningless, since such a state is indeterminable at any individual node of an asynchronous distributed system. The reason for this uncertainty is twofold:

1. There is a non-negligible and unpredictable delay in the transfer of information, including resource allocation state information, from one node (processor) of a distributed system to another. In particular, this delay is large in comparison to local node memory access, so that the transmission delays are large in comparison with the rate at which the system state changes.

2. The possibility of an internode communications transmission error is, while arbitrarily reducible, non-negligible, so that steps must be taken to detect and correct errors during communications. Due to the presence of such errors, it is quite possible that the communications medium will introduce uncertainties with respect to message origination time and sequencing. Attacks on the problem of error handling principally result in the introduction of further (unbounded) delay, as the communication system corrects transmission errors.

The resultant impossibility of ascertaining the global system state of a distributed system clearly renders impossible the globally optimal allocation of system resources to user processes. Even though global optimality cannot be achieved, however, we can, and will, define optimality with respect to whatever information is available. It is this decision making process, then, which we will call *best effort* decision making [Jensen 84].

1.1 Best Effort Decision Making

Best effort decision making cannot be rigorously defined, but we can discuss the concept in some detail. Essentially, it refers to techniques for decision making utilizing incomplete and inaccurate information in the best possible way. Techniques for determining and maximizing the value of the available information, algorithms for cooperative internode decision making, and heuristics for decision search space reduction are among the components of best effort decision making.

It is our thesis that because of the impossiblilty[1] of a globally optimal solution to decentralized resource allocation, resource management decisions should be made using the best information available, making a best effort to determine its meaning. We believe there are several potential techniques (see Section 2.5.1) which can be used to evaluate information in an effort to arrive at a decision once the maximum value has been extracted from the available information. The goal of best effort decision making, in this context, is to make resource allocation decisions as efficiently as possible in the presence of incomplete and inaccurate state information.

A best effort decision, with its prestated globally sub-optimal nature, clearly can be implemented using any of a number of diverse approaches. The approaches to be considered in this research will cover a spectrum including, but not neccessarily limited to, applied AI, decision theory, and non-monotonic logic.

1.2 Real-Time Deadline Management

Such a study will be of limited value, however, in a vacuum, so we will study a particular problem, and several possible best effort decision making approaches will be made to solve it, accompanied by evaluations. The particular problem to be studied will be that of deadline scheduling in a real-time distributed system.

An operating system managing a real-time application shares a large number of functions with a normal timesharing operating system, but differs most significantly in one principal area—the management of time deadlines. The typical time sharing operating system manages a number of independent applications, promoting some concept of fairness (defined by the sytstem administrator) among them. In the real-time system, a single application[2]uses the system resources to solve a single problem, and includes a set of deadlines which must be met in order to satisfy its specifications. We distinguish *hard* real-time systems from *soft* real-time systems in this document; the use of the word *hard* implies that a missed deadline corresponds to a major system failure, while a missed deadline in a *soft* real-time system is not necessarily a major failure, depending on the nature of the deadline, although a specification has been violated. This research will use a computational model encompassing both *soft* and *hard* real-time systems.

The research which we undertake is to study deadline management in a distributed resource sharing environment in which the system state is incomplete and inaccurate; in which the time allocation decisions must be made using a best effort approach. In fact, for this problem a best effort approach must be used because of the built-in uncertainties even if complete and accurate global system state information were available, because of the stochastic nature of the information available about schedulable processes and the limited decision making time available in a real-time system.

---

[1] and intractability, even if global state information were available - see section 2.4

[2] A single application means a set of processes working together and sharing resources toward a common goal. While it is possible for more than one such application to run concurrently in a real-time system, it is usually true that the real-time commitment of the system will be made only to one; any others will execute as background applications.

A considerable amount of research has been done when deadlines could be met, but relatively little information is available about scheduling decisions when available resource limitations require that one or more deadlines cannot be met. Our approach will be designed to maximize the value of the available state information to make the deadline scheduling decisions, particularly in those cases where deadlines cannot be met.

Questions of policy/mechanism separation [Wulf 81] are also related to this problem. If one or more deadlines must be missed, which deadlines should be selected? Clearly a policy decision must be made, and the range of potential policies for this decision will be determined, with the required mechanisms defined, implemented, and evaluated. It is these mechanisms which will reflect the best effort approach, and their support of the potential policies will be analyzed.

2 Motivation

While it is obvious that resource management is *the* fundamental problem in the development of any oerating system, it is equally obvious that this is especially true in a distributed system [Applewhite 82].

The difference between best effort and algorithmic (either consensus or autonomous) decision making in operating system resource management can be illustrated with decisions on the use of resources in a one-person business versus a large corporation. In the small business, resources such as capital and raw materials are all under the control of a single individual, and their current state is generally knowable and therefore manageable by that one individual in an optimum manner. In the large corporation, attempts to completely centralize the resource allocation decisions are generally disastrous. It is well known, however, that the larger business can, at least in principle, make more efficient use of its resources in spite of its seeming disadvantage with respect to the optimality of its resource allocation. Similarly, if we can properly perform best effort resource allocation in a distributed system, the overall utilization of the system's resources should be more effective than in a similar set of uniprocessors.

The question naturally arises: why do almost all existing distributed systems consist of a set of autonomous systems containing communicating processes, in which resources are managed by each node locally for the benefit of its own processes, rather than as a single computational entity, utilizing all resources in the best interests of the system? While there are many answers to this question, the difficulty and importance of decision making is certainly one of the significant reasons. This is exactly the motivation for this research; it is our thesis that, as in the case of the large business, a distributed computer which makes best effort decisions in allocating all system resources for the benefit of the system will be better able to apply those resources to fulfill the user's objectives.

3 Related Research

This area of research is the combination and extension of a number of previous efforts. As a result, there are several areas of research which can be considered related to this work. Here, we will describe research in five related areas:

- *Artificial Intelligence*--particularly techniques for knowledge representation and rules of inference related to decision making (planning).
- *Non-monotonic Logic*—studies related to the handling of multi-valued logic, fuzzy logic (truth assertions with respect to fuzzy sets), and associated truth determination.
- *Decision Theory*—work related to the process of combining sets of observed states of nature, the probabilities associated with these states in the event of a given set of decisions, and the ordering of possible decisions based on this information (e.g., Bayesian theory).
- *Real-Time Deadline Management*—research on deadline scheduling in a real-time system.
- *Distributed Decision Making*—efforts toward the problem of reaching a consensus in a distributed system in the presence of failures.

3.1 Artificial Intelligence

Little or no work has been performed in the application of knowledge collection, knowledge representation, and the resulting inferences to the problem of scheduling, load balancing, or process reconfiguration. It seems that these areas could be critical to the efficient handling of these decisions. Stefik [Stefik 82] delineates a number of approaches to problems involving expert systems, outlining the primary techniques used in developing them based on the overall system type (e.g., planning, diagnostic, etc.). This problem shares a number of the aspects of planning problems, in that the solution must evaluate the data determining its quality and meaning, search the space of possible scheduling decisions estimating the result of each evaluated decsiion, and make a decision which is adequately close to optimal. It is expected that at least some of the techniques involved in the design of expert planning systems will be applicable to the problem of applying a best effort to the real-time process scheduling problem, particularly in the areas of data understanding and knowledge and inference rule representation.

3.2 Non-monotonic Logic

Decision making is dependent on a determination of the state of the relevant world as assessed by the decision maker. The decision to be made can be viewed as a consequence (deduction) from this perceived state, which can be represented as a set of predicates. Normal formal logic, from which inference rules would be taken to produce such deductions, can be described as *monotonic* in the sense that the addition of any new axiom (i.e., an additional observation of state) which is not in conflict with existing axions, can never invalidate a previously drawn conclusion. In a situation characterized by a large quantity of incomplete or inaccurate information, such deductions may later be invalidated by the presence of new information. Logic

in which inferences may be drawn using such data is called non-monotonic logic [McDermott 80][McDermott 82], and is generally characterized by the use of a logical operatior meaning "*is consistent with*" on conclusions drawn.

In another form of monotonic logic, inferences which can be described only as "consistent" with perceived state information can be handled by the use of *fuzzy* sets. Normal set theory deals with sets whose boundaries are clear-cut; an element is either a member of a given set or it is not. Members of fuzzy sets [Kaufmann 75] have an associated membership value which determines the degree of membership of that member in the set. Fuzzy logic refers to the mainipulation of predicates with fuzzy implications (e.g., high value processes should be executed early). Discussions of such logic inference rules are given in [Zadeh 79].

3.3 Decision Theory

There are several mathematical techniques developed to construct decisions based on the opinions (inexact measurements) of a number of co-decision makers. Stankovic [Stankovic 83] describes a heuristic for job assignments in a decentralized, dsitributed environment using Bayesian decision theory [Jeffrey 84], while DeGroot describes iterative solutions given a matrix of opinion weights and a vector of opinions [DeGroot 74].

Bayesian decision theory may be used in an environment in which a decision must be made whose value is determinable only when evaluated in the light of an unpredictable future global state (such as determining whether to plan a picnic for Saturday). The decision maker first constructs a matrix with each row for a potential decision and each column for a potential relevant state, placing in each matrix element a value of the utility or desireability of having made that decision if the corresponding state occurred. Then the decision maker constructs a similar matrix containing the probability that that state will actually occur (possibly conditional on the decision to be made). The row-wise sum of the matrix produced by multiplying each element of the utility matrix by the corresponding element of the probability matrix defines a vector representing the optimal ordering of the set of potential decisions.

3.4 Real-Time Deadline Management

Research into the general scheduling problem has progressed for a long time as a part of operations research, where it has found application in the scheduling of such activities as manufacturing production. Graves [Graves 81] has provided an excellent synopsis of this background, including a taxonomy of production scheduling problems. He identifies four levels of complexity with respect to scheduling decisions; tractable solutions have been identified for only the simplest of these levels while the rest have been proved to be NP-complete or NP-hard.

In particular, the level of complexity most similar to the problem to be considered in this research, soft-real-time deadline scheduling with constant value functions for met deadlines, has been proved to be NP-

comnplete [Karp 72]. Algorithms for this problem have been described [Sahni 76] both for optimum scheduling (obviously, in exponential time) and for a heuristic approach for which an optimal solution can be missed by a determinable factor in $O(n^3)$ time. The model which we are using to define the scheduling problem to be handled includes this problem as a special case.

In a classic paper, Liu and Layland [Liu 73] show that the simple deadline scheduling problem with a set of independent periodic processes whose computation times are exactly known and identical for every period, whose deadlines are equal to their period, and which are running in a single processor, is solvable with a simple algorithm, and that the schedulability of such a simple system is easily determinable in advance. While this result is interesting, the cases covered are much too unrealistic to be directly applicable to our problem.

3.5 Distributed Decision Making

The *Byzantine Generals Problem* [Lamport 82][Dolev 82][Lynch 82] is the name given to the problem of reaching a consensus on a decision in an environment in which some of the decision makers have failed (or may even be antagonistic to the making of a correct decision). It has been proved that in a fully connected network of decision makers [Lamport 82], a correct decision can be effected as long as more than two thirds of the decision makers have not failed. The algorithms presented, however, while guaranteed to produce a correct solution, require full synchronization of all decision makers, and require a large number of messages. It should be noted that in a completely asynchronous system (i.e., one with no complete event ordering such as a clock), no Byzantine agreement is possible in the presence of any failure [Fischer 82].

References

[Allchin 82] Allchin, James E. and Martin S. McKendry.
Object Based Synchronization and Recovery.
Technical Report GIT-ICS-82/15, School of Information and Computer Science, Georgia Institute of Technology, 1982.

[Attar 84] Attar, R., Bernstein P. A. and Goodman N.
Site Initialization, Recovery and Backup in a Distributed Database System.
IEEE Transaction on Software Engineering, Nov., 1984.

[Bentley 83] Bentley, J. L., Johnson, D. S., Leighton, T. and McGeoch, C. C.
An Experiemental Study of Bin Packing.
Technical Report, Bell Laboratories, Murray Hill, N.J. 07974, 1983.

[Bernstein 83] Bernstein, P. A., Goodman, N. and Hadziacos V.
Recovery Algorithms for Database Systems.
Technical Report, Aiken Computation Laboratory, Harvard University, March 1983.

[Chu 80] Chu, W. W.; Holloway, L. J.; Lan, M.; Efe, K.
Task Allocation in Distributed Data Processing.
Computer 13(11):57-69, November, 1980.

[Chvatal 83] Chvatal, V.
Linear Programming.
W. H. Freeman and Company, 1983.

[Coffman 83] Coffman Jr., E. G., Garey, M. R. and Johnson, D. S.
Approximation Algorithms for Bin Packing - An Updated Survey.
Technical Report, Bell Laboratories, Murray Hill, N. J., 1983.

[DeGroot 74] DeGroot, M. H.
Reaching a Consensus.
Journal of the American Statistical Association 69(345):118-121, March, 1974.

[Dolev 82] Dolev, D.
The Byzantine Generals Strike Again.
Journal of Algorithms 3(1):14-30, March, 1982.

[Eswaran 76] Eswaran, K. P., J. N. Gray, R. A. Lorie and I. L. Traiger.
The Notion of Consistency and Predicate Lock in a Database System.
CACM, 1976.

[Fischer 82] Fischer, M. J.; Lynch, N. A.; Paterson, M. S.
Impossibility of Distributed Consensus with One Faulty Process.
Technical Report, Massachusetts Institution of Technology, September, 1982.

[Frederickson 80] Frederickson, G. N.
Probabilistic Analysis for Simple One and Two Dimensional Bin Packing Algorithms.
Information Processing Letters, Vol.11, No. 4, 5., Dec. 1980.

[Garcia-Molina 83]
Garcia-Molina, H.
Using Semantic Knowledge For Transaction Processing In A Distributed Database.
ACM Transaction on Database Systems, Vol 8, No. 2, June, 1983.

[Graves 70] Graves, G. W. and Whinston, A. B.
An Algorithm for The Quadratic Assignment Problem.
Management Science, Vol. 17, No. 7, Mar. 1970.

[Graves 81] Graves, S. C.
A Review of Production Scheduling.
Operations Research 29(4):646-675, July-August, 1981.

[Hillier 80] Hillier, F. S. and Lieberman, G. J.
An Introduction to Operations Research, 3rd Edition.
Holden-Day Inc., 1980.

[Jeffrey 84] Jeffrey, R. C.
The Logic of Decision, Second Edition.
University of Chicago Press, Chicago and London, 1984.

[Jensen 83] Jensen, E. D.
The Archons Project: An Overview.
In *Proceedings of the International Symposium on Synchronization, Control, and Communication in Distributed Systems*, pages 31-35. Academic Press, 1983.

[Jensen 84] Jensen, E. D.
ArchOS: A Physically Dispersed Operating System.
IEEE Distributed Processing Technical Committee Newsletter, June, 1984.

[Kaku 83] Kaku, B. K. and Thompson, G. L.
An Exact Algorithm for The General Quadratic Assignment Problem.
Technical Report, Graduate School of Industrial Administration, Carnegie-Mellon University, Mar. 1983.

[Karp 72] Karp, R. M.
Reducibility among Combinatorial Problems.
Complexity of Computer Computations.
Plenum Press, New York, 1972, pages 397-411.

[Kaufmann 75] Kaufmann, A.
Introduction to the Theory of Fuzzy Sets.
Academic Press, New York, 1975.

[Kleinrock 76] Kleinrock, L.
Queueing Systems, Vol II, pp 424 - 425.
John Wiley and Sons, 1976.

[Korth 83] Korth, H. F.
Locking Primitives in a Database System.
JACM, Vol. 30, No. 1, January, 1983.

[Kung 79] Kung, H. T. and C. H. Papadimitriou.
An Optimal Theory of Concurrency Control for Databases.
In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 116-126. ACM, 1979.

[Lamport 82] Lamport, L.; Shostak, R.; Pease, M.
The Byzantine Generals Problem.
ACM Transactions on Programming Languages and Systems 4(3):382-401, July, 1982.

[Lesser 73] Erman, L. D., Fennell, R. D., Lesser, V. R., and Reddy, D. R.
System Organizations for Speech Understanding.
The Third International Joint Conference on Artificial Intelligence, August 1973.

[Liskov 85] Liskov, B. and Weihl W.
Specification of Distributed Programs.
Technical Report, M.I.T., Sept. 1985.

[Liu 73] Liu, C. L.; Layland, J. W.
Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.
Journal of the Association for Computing Machinery 20(1):46-61, January, 1973.

[Lynch 82] Lynch, N. A.; Fischer, M. J.; Fowler, R. J.
A Simple and Efficient Byzantine Generals Algorithm.
In *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, pages 46-52. ACM-IEEE, July 19-21, 1982.

[Lynch 83] Lynch, N. A.
Multi-level Atomicity - A New Correctness Criterion for Database Concurrency Control.
ACM Transaction on Database Systems, Vol. 8, No. 4, December, 1983.

[Marschak 72] Marschak, J. and Radner, R.
Economic Theory of Teams.
Yale University Press, 1972.

[McDermott 80] McDermott, D.; Doyle, J.
Non-Monotonic Logic I.
Artificial Intelligence 13:41-72, 1980.

[McDermott 82] McDermott, D.
Non-Monotonic Logic II.
Journal of the Association for Computing Machinery 29(1):33-57, January, 1982.

[Mohan 83] Mohan, C.
Efficient Commit Protocol for The Tree Process Model of Distributed Transactions.
IBM Research Report, RJ 3881 (44078), Computer Science, June, 1983.

[Mohan 85] Mohan, C., Fussell, D., Kedem Z. M. and Silberschatz A.
Lock Conversion in Non-Two-Phase Locking Protocols.
IEEE Transaction on Software Engineering, Jan., 1985.

[Papadimitriou 84] Papadimitriou, C. H. and Kanellakis, P. C.
On Concurrency Control by Multiple Versions.
ACM Transaction on Database Systems, Mar., 1984.

[Sahni 76] Sahni, S. K.
Algorithms for Scheduling Independent Tasks.
Journal of the Association for Computing Machinery 23(1):116-127, January, 1976.

[Schwarz 84a] Schwarz, Peter M. and Alfred Z. Spector.
Synchronizing Shared Abstract Types.
ACM Transaction on Computer Systems, AUG, 1984.

[Schwarz 84b] Schwarz, P.
Transactions on Typed Objects.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1984.

[Sha 85a] Sha, L.
Modular Concurrency Control and Failure Recovery -- Consistency, Correctness and Optimality.
PhD thesis, Department of Electrical and Computer Engineering, Carnegie-Mellon University, 1985.

[Sha 85b] Sha, L., Lehoczky, J. P. and Jensen E. D.
Modular Concurrency Control and Failure Recovery --- Part II: Failure Recovery.
Submitted for publication, 1985.

[Sha 85c] Sha, L., Lehoczky, J. P. and Jensen E. D.
Modular Concurrency Control and Failure Recovery --- Part I: Concurrency Control.
Submitted for publication, 1985.

[Silberschatz 80] Silberschatz, A., and Z. Kedem.
Consistency in Hierarchical Database Systems.
JACM 27:1, 1980.

[Stankovic 83] Stankovic, J. A.
Bayesian Decision Theory and Its Application to Decentralized Control of Job Scheduling.
February, 1983.
Internal documentation, University of Massachusetts, Department of Electrical and Computer Engineering.

[Stefik 82] Stefik, M.; Aikins, J.; Balzer, R.; Benoit, J.; Birnbaum, L.; Hayes-Roth, F.; Sacerdoti, E.
The Organization of Expert Systems, A Tutorial.
Artificial Intelligence 18:135-173, 1982.

[Svobodova 84] Svobodova, L.
Resilient Distributed Computing.
IEEE Transaction on Software Engineering, May, 1984.

[Tokuda 85] Tokuda, H., Clark, R. K. and Locke, C. D.
Archons Operating System (ArchOS) --- Client Interface, Part II.
Technical Report, Department of Computer Science, Carnegie-Mellon University, 1985.

[Weihl 84] Weihl, W. E.
Specification and Implementation of Atomic Data Types.
PhD thesis, Massachusetts Institute of Technology, 1984.

[Wulf 81] Wulf, W. A.; Levin, R.; Harbison, S. P.
HYDRA/C.mmp: An Experimental Computer System.
McGraw-Hill, Inc., 1981.

[Zadeh 79] Zadeh, L. A.
A Theory of Approximate Reasoning.
Machine Intelligence 9.
Wiley, New York, 1979.

4.2 Best Effort Decision Making Theory

Best Effort Decision Making Theory

1 Introduction

In a decentralized computer system, processes often function co-operatively as a team to enhance system level performance. One basic goal of the Archons project of which this research is part is to develop concepts and mechanisms which are intended to function co-operatively by using co-operative decision making. These mechanisms must function efficiently even though they will have both inaccurate and incomplete information upon which to act [jensen84]. In this chapter, we introduce a framework for dealing with co-operative decision making problems arising in decentralized systems in general and the Archon system specifically. Concepts drawn from queueing theory, team decision theory [Marschak72] and information theory will be used to analyze the value of system status information for resource management. We focus specifically on the trade-offs between the completeness and the timeliness of information. Rather than treating the consensus problem abstractly, we will single out a particular problem for study: the problem of decentralized load balancing. This problem will nicely illustrate the application of one particular paradigm of consensus decision making – team decision theory. Furthermore, it illustrates the trade-offs among the cost of information (loading information at each processor), its completeness, and its timeliness. This chapter is organized as follows. Section . 2 provides an overview of decentralized decision making. Section · 3 describes the load balancing problem and its formulation as a consensus decision making problem. Specific results characterizing the value of system status information are derived. Section 4 presents some ideas for further study.

2 Overview of Co-operative Decision Making

In decentralized computer systems, the processors often must collaborate to solve a particular system or application problem. The problems may range from those associated with the management of system resources (such as deciding which processor should process a particular task) to those arising in a specific application such as speech recognition. We propose to try to deal with these problems in a general way by casting them in a framework of statistical decision theory. Once in such a framework, it is possible to consider issues such as the importance or value of information and its decline in value with time delays.

In statistical decision theory, there is a set of possible states one of which is the "true" state. Generally, it is desired to identify the particular true state. Often data are available to assist the decision maker in identifying the true state. These data have different likelihoods for the various possible states. An example might be the load balancing problem in a distributed local computer network. The possible states represent the exact workloads at each of the processors at the time the decision is made. The data are the workload information

messages exchanged by the processors. These messages are inexact and delayed in time. The particular decision made will vary in goodness with the possible states of the system.

Many decision problems arising in decentralized decision making can be cast in a statistical decision theory framework. Once in this framework, various solution techniques can be brought to bear on the problems. A particularly powerful method, Bayesian decision theory, should be considered, since it leads to optimal decision procedures. In the Bayesian formulation, the probability distribution is specified on the possible states. This is called a prior distribution. Next, we must determine the probability of the possible states with respect to observed data, which is called the likelihood function. These two components result in a posterior distribution over the possible states. The posterior distribution reflects both the prior information and the observed data. An optimal decision or action with respect to some criterion can then be determined. Furthermore, the posterior distribution can be continuously updated in real-time. Unfortunately, this program can often not be carried out in its exact form. Two fundamental difficulties arise. First, the prior distribution and likelihood function may be essentially impossible to specify. Second, in many examples the Bayesian formulation does not consider that data (for example processor status information) deteriorates in value with time. If some information is delayed, the quality of the decision made may be greatly reduced.

In many real problems, a joint likelihood function must be determined empirically. When a high dimension likelihood function is required, an empirical approach is impossible to carry out. For example, in a speech recognition context, the likelihood of a particular signal being a particular phrase will depend on many factors ranging from its context in the sentence to the sound of the actual signal. A speech recognition system will take these many diverse factors into account. The Bayesian program would call for these factors to be considered jointly rather than marginally (separately). Currently, this is an impossible task. A satisfactory but sub-optimal approach is to employ a team of specialized processors. Each processor works on a special aspect of the problem. Each processor possesses the special database and codes designed for its particular aspect of the problem. The individual processors are to function as a team of experts and are to arrive at a consensus decision. Each processor develops hypotheses and probabilities associated with the hypotheses. The team members exchange these hypotheses and by this process enhance the marginal viewpoints to a more global view. The process must somehow converge to a consensus decision of the team perhaps by a formal method such as that of DeGroot [DeGroot74] or by some heuristic method such as the HearSay system [Lesser73]. The approach is somewhat similar in spirit to the "divide and conquer" approach of algorithm design but lacks the optimality of the full Bayesian program, because the joint information has been sacrificed. That is, the inter-dependence of various aspects are only approximated by the indirect approach of reaching a consensus among experts. In the following discussion, we refer to this type of co-operative decision making process as consensus decision making.

A second consideration which often leads to decentralized decision making is the delay in and the cost of interprocess communication. In many real-time applications such as load balancing, information delay often

plays a dominant role, and rapid decision making may be crucial. The time delay incurred in collecting complete status information from all the relevant parties could be sufficiently long that by the time all the information is gathered and a decision is made, valuable time will have been lost, and the information itself will be substantially in error. To obtain a timely decision, the decision makers may be forced to make decisions based only on partial information. For example, in a large point-to-point network, a dynamic routing scheme is needed for message transfers through the network. Clearly, elaborate information gathering and consensus arbitration is self defeating. Instead, one might adopt decentralized procedures such as the one used in Arpanet [Kleinrock76] and allow each node to make its own local routing decisions. Furthermore, each node will make these decisions based on only partial information, the information from nearby nodes. Information from more distant nodes will be significantly delayed and of lesser value. This illustration highlights the importance of the two concepts of timeliness and completeness of information. Since this type of co-operative decision making requires decision makers to work as a team to further the system level performance but does not require them to reach an agreed upon opinion, we will refer to this type of co-operative decision making as team decision making [Marschak72]. A single team member is allowed to make a certain set of decisions without necessarily gaining the concurrence of the other team members. Any team member may seek information which will improve the quality of its decisions. All team members make decisions which attempt to improve overall system performance.

The consensus decision making procedure in which a single decision is reached, and the team decision procedure can be thought of as extremes on a negotiation spectrum. Generally, there are several important facets in any consensus decision making problem which will, in part, determine the most appropriate solution scheme. These include:

- Is it necessary to reach a single consensus among all decision makers? Can a single decision maker make the decision after taking the opinions and information of others into account?
- How important is the completeness of the information in making an optimal or near optimal decision? That is, how does the quality of a decision deteriorates as the completeness of information decreases?
- What is the computational complexity of the proposed decision process? Does a substantial reduction in complexity cause a major or minor reduction in performance?
- What is the total delay of a proposed decision process? This includes the information gathering, negotiation, and decision making. More importantly, how is this total delay compared with the relaxation time of the system? The relaxation time is a measure of how fast the system is changing its status.

In the following, we attempt to answer some these questions in the context of load balancing.

3 The Load Balancing Problem

In this section, we consider the load balancing problem in the context of a local distributed computer system such as Archons [Jensen83] in which users create jobs at the nodes they log into. Since both the job arrival process and the required computational time for each node is unpredictable, the loads at each of the nodes will be widely variable. Some will be lightly loaded, while others will be heavily loaded. The task of the decision makers (local operating systems) is to equalize the loads in the system in order to improve the system wide performance. Balanced loads will create short queues and short response times.

Each decision maker can manage its own job queue if it chooses to process it locally. Load balancing necessitates the shifting of load from one processor to another, and this necessitates some form of consensus and negotiation involving at least a subset of the processors. At the other end of the spectrum, one processor acting as a central scheduler will assign the job to one of the processors. The no negotiation approach can be carried further by letting each of the processors schedule its own jobs. This arrangement would then be equivalent to the team decision approach whereby no consensus is needed, but the individual decision makers act to improve overall system performance. There is no a priori necessity for consensus. Rather, the degree of negotiation must be dictated by performance.

Generally, there are two aspects of load balancing. The first aspect is the matching of the structure of the data flow in the application with the architecture of the distributed computer system. This includes clustering closely coupled processes in the same or nearby nodes, scheduling tasks with respect to precedence relations in such a way that more tasks can be executed concurrently, and matching the special resource requirements of tasks with special hardware etc. From a modelling point of view, the optimization of these aspects is typically an integer programming problem. In many dedicated real-time control systems, this is done off-line due to the lack of efficient algorithms for solving integer programming problems. A second aspect of load balancing is the control of system dynamics. In a distributed system, there is substantial redundancy built into the system. Often a class of tasks can be executed with near equal efficiency by any of a group of similar or identical processors. The problem then is the equalization of the load so as to improve the throughput and response times. In this chapter, we will concentrate on this latter aspect.

Traditionally, the load balancing is carried out by a centralized scheduler. However, this approach is inadequate for many larger scale systems which are typically loosely coupled. The inadequacy is in part due to reliability considerations and in part due to the time delay involved in routing all relevant information to a centralized location. In this chapter, we will investigate a highly parallel and highly decentralized approach. In this approach, each processor is considered to be a member of a management team, the individual actions of which are geared to increasing the system level performance. The crucial requirement is the development of an information exchange structure and an associated decision procedure which makes such an approach effective.

In summary, it appears that one reasonable approach to load balancing is the highly decentralized team approach. In this approach, each processor acts individually to increase system performance. The key ingredient is to develop an information exchange structure which will lead to increased system performance by allowing each individual processor to schedule its own jobs. The most difficult aspect lies in obtaining the relevant system status information in a timely fashion. In addition, the actions of the individual processors must be sufficiently coordinated to generate a coherent global strategy, even though it is implemented in a highly decentralized fashion. We present some results on this team formulation in the rest of this section.

3.1 The Value Of Perfect Information

In developing a highly decentralized team approach to the load balancing problem, an important consideration is the value of system status information. The actual information available to a processor will often be delayed, incomplete, and inaccurate. It is, however, useful to consider the ideal case of "perfect" information. System performance measures such as throughput or mean response time are influenced by unavoidable queueing or congestion delays and delays caused by imperfect system status information. To quantify the importance of system status information, we must separate out the extra delay incurred when information is imperfect. The approach is to measure system performance under the assumption of perfect information and to also measure it under some imperfect information scheme. The former gives an upper bound on obtainable performance, while the difference measures the loss in performance due to imperfect information.

As an illustration, let us contrast two extreme cases. At one extreme, we assume that all processors have complete, accurate, and instantaneous system information. At the other extreme, we assume that processors have no system status information at all (not even local information). In both cases, the processors will make the best possible decisions, but since different information levels are assumed, the performance achieved in the two situations will be different. The difference in performance is due solely to the availability of the status information and provides us with a quantitative measure of the value of complete information. This measure will help us to identify situations in which information is very valuable and situations where it is not.

As a specific example of this approach, we assume that the network consists of n homogeneous processors. We assume jobs arrive at the i th according to a Poisson process with mean λ_i jobs per unit time. All jobs are homogeneous and require a random amount of time to process given by an exponential distribution with mean m. We will, for the moment, ignore all other factors such as communication costs, priorities, etc. In the case of perfect information, each processor is able to act as a central scheduler. If a processor is available for work, this is assumed to be known to the other processors. No queues will form if other processors are idle. This corresponds to an M/M/n queueing system with arrival rate $\lambda = \lambda_1 + \ldots + \lambda_n$ and mean service rate 1/m. The performance of such a system can be explicitly characterized. For example, the mean response time is given by $n\rho((1-\rho)+C(n,n\rho)/n)/(\lambda(1-\rho))$ where C is the Erlang C function and $\rho = \lambda m$. The Erlang C function is bounded above by 1 and is usually small. Thus this formula can be approximated by $n\rho/\lambda$. Other quantities such as processor utilization can also be calculated.

In the case of no system status information, each processor must make a decision based only on the λ_i, m, and n. That is, only long run average information rather than dynamic loading information is available to the processors in this case. The optimal assignment scheme is probabilistic and results in an even load being put on all processors. It is important to observe that the concept of a balanced load is defined here only in the long run. All jobs are allocated so that in the long run the same number of jobs will be handled by all n processors. The lack of current and complete system status information means that the processors are not able to take current loads into account. Imbalances will result, and system performance will be decreased.

The situation of no current status information corresponds to a collection of n independent M/M/1 queueing systems with Poisson input having mean rate λ/n and mean service time m. Performance quantities are easily calculated for such a system. For example, the mean response time is given by $n\rho/(\lambda(1-\rho))$. It is interesting to compare the mean response time in these two extreme cases, the complete information case and the no information case. The two differ by the multiplicative factor $1-\rho + C(n,n\rho)/n$. One might refer to this quantity as the "information factor". The factor is equal to 1, its minimum value, when $\rho=0$, and it decreases to $1/n$ as ρ increases to 1. This means that at low traffic intensities (ρ near 0), system status information provides no significant reduction. This factor can be calculated for any traffic intensity ρ and shows that information is very valuable at moderate and high intensities. Furthermore, the value of information increases with the number of processors in the network.

3.2 The Value Of Local Information

In this section, we continue to assume a situation in which processes and processors are homogeneous. Furthermore, we do not consider the communication delays or the time cost of transferring tasks in this section. Now we wish to study the situation in which each processor has only local information (the status of its own job queue) and knows the long run arrival rate of work. The arrivals are assumed to be independent Poisson processes with common parameter λ, while service times are again exponential with mean m. Each processor upon receiving a job must determine whether to process it locally or send it to another processor. This decision must be based only on the local queue length and is made without any negotiation. We also assume that if it is sent to another processor, that processor must accept the job and cannot send it further. This assumption is useful in that it prevents excessive job swapping, and it simplifies the analysis.

A general class of control policies within which an optimal policy must lie consists of a set of probabilities, $\{p_n\}$, where p_n gives the probability that the processor accepts a newly arrived task when its local queue consists of n other tasks. These probabilities must be decreasing in n. Optimal stochastic control theory suggests that the optimal p_ns will be non-randomized (either 0 or 1). This reduces consideration to a 'control *limit' policy*: *accept a task if and only if the local queue, including the one being served, consists of L or* fewer tasks, otherwise send it to another processor chosen from among the other candidates. The best value of L will depend on the traffic intensity. If the traffic intensity is small, then L = 1 is appropriate. For larger values of ρ, L must be larger. It remains to determine the optimum value of L for each ρ, and to assess the performance of the resulting local information system.

Once L has been determined, there is a second phase to the policy - the choice of the processor to send the unaccepted task. A number of policies, both deterministic and stochastic can be used. An optimal policy must equalize the load on the processors in the long run. Beyond that requirement, we expect that the optimal policy will minimize the coefficient of variation of the resulting arrival processes. The form of the optimal policy is still an open question. We choose a policy in which each processor selects randomly from the remaining $(n-1)$ available choices.

The L policy queueing system has not been studied before, and it is quite complicated to produce exact analytic solutions. Fortunately, a simple approximation can be used which is very accurate for large values of $n(1-\rho)$. We treat each of the n processors as independent birth-death processes with birth rate $b_n = \lambda(1+s)$ for $n \leq L$, and $b_n = \lambda s$ for $n > L$, while the death rate is $1/m$ for all states. The parameter s is used to connect the queues together and is chosen to achieve equilibrium. The unknown s is a root of a polynomial equation. Once s has been determined, approximate equilibrium distribution and its mean can be found. The derivation is given in the appendix along with simulation results which attest the high accuracy of the approximation. If we let $F = n\rho/(\lambda(1-\rho))$, then we can determine the mean response times to be given by

POLICY	MEAN RESPONSE TIME
no information	$F[1]$
local control, L=1	$F[1/(1+\rho)]$
local control, L=2	$F[1-\rho+\rho/((1+\rho)^2)]$
perfect information	$F[1-\rho+C(n,n\rho)/n] \sim F[1-\rho]$

The results are rather surprising. The percentage reduction in mean response time for perfect information over no information is ρ. The percentage reduction for the $L=1$ policy is $\rho/(1+\rho)$. Of the total ρ percent reduction for perfection, $1/(1+\rho)$ of it can be obtained using only local information. Even at high intensities, local information gives over 50% of the possible gain. If one allows general L's, then at least 62% is possible for any traffic intensity! It is worthwhile to point out that some non-local information can be obtained at nearly no additional cost by keeping track of the source of each task in the local queue. Hence one can further improve system performance by using this "free" non-local information. For example, if a processor decides to ship out a task, it should be sent to a processor which has not shipped it anything in its current queue. This all suggests that very effective control can be exercised at low traffic intensities using just local control. At higher intensities, the mean response times become quite long. Even though local control can capture a high percentage of the overall gain possible from perfect information, it becomes important to refine the decision making with extra information. Specifically, one needs to make a more informative choice of the processor selected to process the task. The purely random choice described earlier is inadequate. Several such schemes are suggested in the next section.

3.3 Approaches to Decentralized Load Balancing

Once one has introduced a measure of the value of information, it remains to develop an algorithm which will nearly achieve the theoretical upper bound on performance. In general, processors will send messages to inform other processors of their status. The more frequent the messages, the more information other processors will have. Unfortunately, these messages create overhead which serves to degrade system performance. To deal with this tradeoff, one can draw an analogy with team decision theory. Information is valuable and worth the cost only if it causes the recipient to change his decision and results in a lower overall cost. A message should be sent only if it has a sufficiently large information content to justify the cost of sending it. Here we are ignoring the reliability considerations which may dictate that a message be sent periodically. We use the word 'information' in the Shannon sense. The information gained from a message is equal to the uncertainty removed. Only messages describing relatively unusual or surprising system states should be transmitted. In the load balancing context, the high information content messages are those which describe overloading or underloading conditions. This suggests that processors should transmit status messages according to a control limit policy: either when it is underloaded or overloaded relative to its equilibrium distribution. The exact control limits are determined to optimize the tradeoff between the cost of information and the gain in performance from it.

The control limit policy described in the previous paragraph should provide a very efficient decentralized control algorithm. The policy is, however, one based on long run equilibrium calculations which ignore short run fluctuations. A processor will send a status message when its workload deviates from its long run expectations. It is possible that there could be a heavy influx of jobs over a short period (or a very small influx), and many processors will have above (below) average workloads. This causes many status messages to be sent which further degrades the system. This effect can be overcome by having control limits which are determined dynamically and change based on short term load fluctuations. Unfortunately, this would seem to necessitate sending even more messages to identify these short run fluctuations. There is, however, an alternative approach. This approach is based on having each processor construct a system status model. If we assume that the processors are linked by a bus connection, then each processor must do intensive bus monitoring. Each processor must keep track of all traffic on the bus, noting its source, its destination, and if possible its estimated processing time. This information allows each processor to build and update a system status model. Such a model would predict the current workload at each processor. Based on this model, the processor could act as a central scheduler to determine the best processor to handle any particular task. The quality of these decisions will be totally a function of the accuracy of the model used. At first glance, it would seem that the system status model might be accurate for a short time period, but its quality would quickly deteriorate as time passes, due in part to the uncertainty of the exact processing requirements of any particular task. It is necessary to introduce some sort of feedback control to keep the model accurate. This can be also done in a highly decentralized efficient manner. Under the assumption that the processors are coupled by a bus, they will all see the same traffic. As a result, they will all build identical system status models. Thus the models used by each of the processors gives essentially identical predictions of the

workload at any particular processor. In addition, each of the processors has extra information in that it knows the exact workload for itself. Each processor can compare its own workload with the workload as predicted by the common system status model. The two will of course deviate. It is only important to notify the other processors when this deviation becomes significant. This is again done on a control limit basis. When the actual load is significantly larger or smaller than the load predicted by the model, that processor must send a message to all other processors to have that part of the model updated. The exact control limits must be determined to tradeoff the increase in performance with the overhead costs in sending the message. In this fashion an effective feedback control mechanism can be established in a highly decentralized fashion.

4 Future Research

The previous section indicates how one can quantify the value of system status information. We wish to generalize these results to a broader context. This will be done as follows. We will assume that both jobs and processors may be heterogeneous. Further we will reintroduce the costs (in time) of communication associated with sending jobs around the network. A number of models are possible to handle this greater degree of generality. We might assume that processes arriving at processor i (at rate λ_i) can be processed by any other processor, but the amounts of time required will vary. A number of factors will cause these time differences: differing communication times, the capabilities of the particular processors, the requirements of the job, the location of the data, etc. We assume that the time to process a job at processor j is random and has a distribution F_{ij} with mean m_{ij} and variance s_{ij}. From this structure, one must evaluate the system performance under a full information and a no information structure.

Under the assumption of no dynamic information, the decisions must be based solely on the parameters λ_i and F_{ij}. Again a probabilistic algorithm will be optimal. Performance quantities such as the mean response time can be calculated from the Pollaczek-Khinchin formula. The full information optimal performance evaluation can be carried out by a straightforward Lagrange multipliers argument.

It is intuitively clear that heterogeneity in general reduces the value of system status information. This observation follows because knowledge of the availability of a particular processor will be useless for jobs which are mismatched to that processor. Heterogeneity increases the number of such mismatches and thus reduces the number of cases where information is of value.

In summary, there remain a variety of important issues yet to be investigated. First, there is a need to evaluate the response times of systems more general than the one analyzed in sections 1 and 2. This is outlined above. Second, one must evaluate the performance of the various control strategies outlined in section 3.3. These initial results do, however, indicate that the team decision approach with limited local information will offer a very high performance decentralized system.

5 Appendix

The behavior of the network under an L policy can be determined approximately using a simple birth and death process model. We focus attention onto a single node. This node receives external input according to a Poisson (λ) process. It accepts and queues these tasks if there are currently fewer than L tasks there. It sends the work elsewhere if it currently has L or more tasks. The node also accepts tasks sent by other nodes that invoked the L policy. These tasks must be kept and processed. That is, each task can be only sent once by a node. We assume the same external input rate λ and exponential service rate 1/m at each node.

The queue length process at each node is treated as a birth and death process with constant death rate 1/m. The birth rate is $b_i = \lambda + \lambda\theta = \lambda(1+\theta)$ if $0 \le i < L$ and is $\lambda\theta$ if $i \ge L$. Here θ is the fraction of traffic forwarded by a node and is $\sum_{j=L}^{\infty} \pi_j$, where $\{\pi_k\}_{k=0}^{\infty}$ is the equilibrium distribution of the birth-death process. One can find the equilibrium distribution from standard birth and death theory.

$$\pi_k = \pi_0 \prod_{i=0}^{k-1}(b_i m_{i+1}), \quad 1 \le k < \infty$$

$$\sum_{k=0}^{\infty} \pi_k = 1.$$

Consequently,

$$\pi_k = \begin{cases} \rho^k(1+\theta)^k & 1 \le k \le L \\ \rho^k\theta^{k-L}(1+\theta)^L & k > L \end{cases}$$

3.5.1 Case 1: L = 1

We consider the case L = 1. Here, the equilibrium distribution is given by

$$\pi_k = \begin{cases} \rho(1+\theta)\pi_0 & k=1 \\ \rho^k\theta^{k-1}(1+\theta)\pi_0 & k>1 \end{cases}$$

where $\theta = 1 - \pi_0$, $1 = \sum_{j=0}^{\infty} \pi_j$,

The condition $1 = \sum_{j=0}^{\infty} \pi_j$ yields $\pi_0(1+\rho)/(1-\rho\theta) = 1$. The further requirement $\theta = 1 - \pi_0$ gives $\pi_0 = 1 - \rho$, so $\theta = \rho$. This gives the equilibrium distribution in terms of ρ alone,

$$\pi_k = \begin{cases} 1-\rho & k=0 \\ \rho^{2k-1}(1+\rho)(1-\rho) & k \ge 1 \end{cases}$$

The mean queue length can be calculated directly from this distribution to be $\rho^2/[(1-\rho)^2(1+\rho)]$. Little's formula can be used to find the waiting time = $F(1/(1+\rho))$.

5.2 Case 2: L = 2

The analysis for L = 2 is similar to the L = 1 case. The equilibrium distribution is given by

$$\pi_k = \begin{cases} \rho(1+\theta)\pi_0 & k=1 \\ \rho^k\theta^{k-2}(1+\theta)^2\pi_0 & k\geq 2 \end{cases}$$

where $\sum_{i=0}^{\infty}\pi_i = 1$, $\theta = 1-\pi_0-\pi_1$.

The three conditions, $\sum_{i=0}^{\infty}\pi_i = 1$, $\theta = 1-\pi_0\pi_1$ and $\pi_1 = \rho(1+\theta)\pi_0$ can be used to find the equilibrium distribution. It is $\pi_0 = 1-\rho$ and

$$\pi_k = \begin{cases} \rho(1+\rho)(1-\rho)/(1+\rho(1-\rho)) & k=1 \\ [\rho^3/(1+\rho(1-\rho)]^k(1+\rho)^2(1-\rho)/\rho^4 & k\geq 2 \end{cases}$$

Once again, the equilibrium queue length can be found this distribution. The waiting time is given by $F(1-\rho+\rho/(1+\rho)^2)$

5.3 Simulation Results

It is important to determine the accuracy of the previous approximation. A simulation was carried out for that purpose. Selected results are presented below for the case of 5 nodes. The results show that the approximation is extremely accurate for traffic intensities up to 0.5 and still accurate when traffic intensity is 0.7. At high traffic intensities, the network becomes difficult to simulate accurately as extremely long runs are needed. The approximation will work better as the number of nodes increases, but simulations are difficult to carry out in such cases. The followings are simulation results and theoretical results of the probability of n jobs in the queue. In addition, the mean queue length of simulation results and theoretical results are also compared.

Run 1: Traffic Intensity = 0.1, Policy: L = 1

	Simulation	Theory
Prob[n=0]	0.902	0.900
Prob[n=1]	0.097	0.099
Prob[n=2]	0.001	0.001
Prob[n≥3]	0.000	0.000
Mean queue length	0.099	0.101

Run 2: Traffic Intensity = 0.1, Policy: L = 2

	Simulation	Theory
Prob[n=0]	0.901	0.900
Prob[n=1]	0.090	0.091
Prob[n=2]	0.009	0.009
Prob[n≥3]	0.000	0.000
Mean queue length	0.108	0.109

Run 3: Traffic Intensity = 0.3, Policy: L = 1

	Simulation	Theory
Prob[n=0]	0.702	0.700
Prob[n=1]	0.268	0.273
Prob[n=2]	0.026	0.025
Prob[n=3]	0.003	0.002
Prob[n≥4]	0.001	0.000
Mean queue length	0.333	0.329

Run 4: Traffic Intensity = 0.3, Policy: L = 2

	Simulation	Theory
Prob[n=0]	0.699	0.700
Prob[n=1]	0.225	0.226
Prob[n=2]	0.074	0.073
Prob[n=3]	0.002	0.001
Prob[n≥4]	0.000	0.000
Mean queue length	0.379	0.375

Run 5: Traffic Intensity = 0.5, Policy: L =1

	Simulation	Theory
Prob[n=0]	0.502	0.500
Prob[n=1]	0.365	0.375
Prob[n=2]	0.094	0.094
Prob[n=3]	0.027	0.023
Prob[n=4]	0.008	0.006
Prob[n=5]	0.003	0.002
Prob[n≥6]	0.001	0.000
Mean queue length	0.687	0.666

Run 6: Traffic Intensity = 0.5, Policy: L = 2

	Simulation	Theory
Prob[n=0]	0.502	0.500
Prob[n=1]	0.292	0.300
Prob[n=2]	0.179	0.180
Prob[n=3]	0.022	0.018
Prob[n=4]	0.004	0.002
Prob[n≥5]	0.001	0.000
Mean queue length	0.737	0.722

Run 7: Traffic Intensity = 0.7, Policy: L = 1

	Simulation	Theory
Prob[n=0]	0.299	0.300
Prob[n=1]	0.344	0.357
Prob[n=2]	0.167	0.175
Prob[n=3]	0.087	0.086
Prob[n=4]	0.046	0.042
Prob[n=5]	0.025	0.021
Prob[n=6]	0.014	0.010
Prob[n=7]	0.008	0.005
Prob[n≥8]	0.010	0.004
Mean queue length	1.468	1.365

Run 8: Traffic Intensity = 0.7, Policy: L = 2

	Simulation	Theory
Prob[n=0]	0.301	0.300
Prob[n=1]	0.279	0.295
Prob[n=2]	0.273	0.290
Prob[n=3]	0.086	0.082
Prob[n=4]	0.034	0.023
Prob[n=5]	0.015	0.007
Prob[n=6]	0.007	0.002
Prob[n≥7]	0.005	0.001
Mean queue length	1.371	1.267

References

[Allchin 82] Allchin, James E. and Martin S. McKendry.
Object Based Synchronization and Recovery.
Technical Report GIT-ICS-82/15, School of Information and Computer Science, Georgia Institute of Technology, 1982.

[Attar 84] Attar, R., Bernstein P. A. and Goodman N.
Site Initialization, Recovery and Backup in a Distributed Database System.
IEEE Transaction on Software Engineering, Nov., 1984.

[Bentley 83] Bentley, J. L., Johnson, D. S., Leighton, T. and McGeoch, C. C.
An Experiemental Study of Bin Packing.
Technical Report, Bell Laboratories, Murray Hill, N.J. 07974, 1983.

[Bernstein 83] Bernstein, P. A., Goodman, N. and Hadziacos V.
Recovery Algorithms for Database Systems.
Technical Report, Aiken Computation Laboratory, Harvard University, March 1983.

[Chu 80] Chu, W. W.; Holloway, L. J.; Lan, M.; Efe, K.
Task Allocation in Distributed Data Processing.
Computer 13(11):57-69, November, 1980.

[Chvatal 83] Chvatal, V.
Linear Programming.
W. H. Freeman and Company, 1983.

[Coffman 83] Coffman Jr., E. G., Garey, M. R. and Johnson, D. S.
Approximation Algorithms for Bin Packing - An Updated Survey.
Technical Report, Bell Laboratories, Murray Hill, N. J., 1983.

[DeGroot 74] DeGroot, M. H.
Reaching a Consensus.
Journal of the American Statistical Association 69(345):118-121, March, 1974.

[Dolev 82] Dolev, D.
The Byzantine Generals Strike Again.
Journal of Algorithms 3(1):14-30, March, 1982.

[Eswaran 76] Eswaran, K. P., J. N. Gray, R. A. Lorie and I. L. Traiger.
The Notion of Consistency and Predicate Lock in a Database System.
CACM, 1976.

[Fischer 82] Fischer, M. J.; Lynch, N. A.; Paterson, M. S.
Impossibility of Distributed Consensus with One Faulty Process.
Technical Report, Massachusetts Institution of Technology, September, 1982.

[Frederickson 80] Frederickson, G. N.
Probabilistic Analysis for Simple One and Two Dimensional Bin Packing Algorithms.
Information Processing Letters, Vol.11, No. 4, 5., Dec. 1980.

[Garcia-Molina 83]
Garcia-Molina, H.
Using Semantic Knowledge For Transaction Processing In A Distributed Database.
ACM Transaction on Database Systems, Vol 8, No. 2, June, 1983.

[Graves 70] Graves, G. W. and Whinston, A. B.
An Algorithm for The Quadratic Assignment Problem.
Management Science, Vol. 17, No. 7, Mar 1970.

[Graves 81] Graves, S. C.
A Review of Production Scheduling.
Operations Research 29(4):646-675, July-August, 1981.

[Hillier 80] Hillier, F. S. and Lieberman, G. J.
An Introduction to Operations Research, 3rd Edition.
Holden-Day Inc., 1980.

[Jeffrey 84] Jeffrey, R. C.
The Logic of Decision, Second Edition.
University of Chicago Press, Chicago and London, 1984.

[Jensen 83] Jensen, E. D.
The Archons Project: An Overview.
In *Proceedings of the International Symposium on Synchronization, Control, and Communication in Distributed Systems*, pages 31-35. Academic Press, 1983.

[Jensen 84] Jensen, E. D.
ArchOS: A Physically Dispersed Operating System.
IEEE Distributed Processing Technical Committee Newsletter, June, 1984.

[Kaku 83] Kaku, B. K. and Thompson, G. L.
An Exact Algorithm for The General Quadratic Assignment Problem.
Technical Report, Graduate School of Industrial Administration, Carnegie-Mellon University, Mar. 1983.

[Karp 72] Karp, R. M.
Reducibility among Combinatorial Problems.
Complexity of Computer Computations.
Plenum Press, New York, 1972, pages 397-411.

[Kaufmann 75] Kaufmann, A.
Introduction to the Theory of Fuzzy Sets.
Academic Press, New York, 1975.

[Kleinrock 76] Kleinrock, L.
Queueing Systems, Vol II, pp 424 - 425.
John Wiley and Sons, 1976.

[Korth 83] Korth, H. F.
Locking Primitives in a Database System.
JACM, Vol. 30, No. 1, January, 1983.

[Kung 79] Kung, H. T. and C. H. Papadimitriou.
An Optimal Theory of Concurrency Control for Databases.
In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 116-126. ACM, 1979.

[Lamport 82] Lamport, L.; Shostak, R.; Pease, M.
The Byzantine Generals Problem.
ACM Transactions on Programming Languages and Systems 4(3):382-401, July, 1982.

[Lesser 73] Erman, L. D., Fennell, R. D., Lesser, V. R., and Reddy, D. R.
System Organizations for Speech Understanding.
The Third International Joint Conference on Artificial Intelligence, August 1973.

[Liskov 85] Liskov, B. and Weihl W.
Specification of Distributed Programs.
Technical Report, M.I.T., Sept. 1985.

[Liu 73] Liu, C. L.; Layland, J. W.
Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.
Journal of the Association for Computing Machinery 20(1):46-61, January, 1973.

[Lynch 82] Lynch, N. A.; Fischer, M. J.; Fowler, R. J.
A Simple and Efficient Byzantine Generals Algorithm.
In *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, pages 46-52. ACM-IEEE, July 19-21, 1982.

[Lynch 83] Lynch, N. A.
Multi-level Atomicity - A New Correctness Criterion for Database Concurrency Control.
ACM Transaction on Database Systems, Vol. 8, No. 4, December, 1983.

[Marschak 72] Marschak, J. and Radner, R.
Economic Theory of Teams.
Yale University Press, 1972.

[McDermott 80] McDermott, D.; Doyle, J.
Non-Monotonic Logic I.
Artificial Intelligence 13:41-72, 1980.

[McDermott 82] McDermott, D.
Non-Monotonic Logic II.
Journal of the Association for Computing Machinery 29(1):33-57, January, 1982.

[Mohan 83] Mohan, C.
Efficient Commit Protocol for The Tree Process Model of Distributed Transactions.
IBM Research Report, RJ 3881 (44078), Computer Science, June, 1983.

[Mohan 85] Mohan, C., Fussell, D., Kedem Z. M. and Silberschatz A.
Lock Conversion in Non-Two-Phase Locking Protocols.
IEEE Transaction on Software Engineering, Jan., 1985.

[Papadimitriou 84] Papadimitriou, C. H. and Kanellakis, P. C.
On Concurrency Control by Multiple Versions.
ACM Transaction on Database Systems, Mar., 1984.

[Sahni 76] Sahni, S. K.
Algorithms for Scheduling Independent Tasks.
Journal of the Association for Computing Machinery 23(1):116-127, January, 1976.

[Schwarz 84a] Schwarz, Peter M. and Alfred Z. Spector.
Synchronizing Shared Abstract Types.
ACM Transaction on Computer Systems, AUG, 1984.

[Schwarz 84b] Schwarz, P.
Transactions on Typed Objects.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1984.

[Sha 85a] Sha, L.
Modular Concurrency Control and Failure Recovery -- Consistency, Correctness and Optimality.
PhD thesis, Department of Electrical and Computer Engineering, Carnegie-Mellon University, 1985.

[Sha 85b] Sha, L., Lehoczky, J. P. and Jensen E. D.
Modular Concurrency Control and Failure Recovery --- Part II: Failure Recovery.
Submitted for publication, 1985.

[Sha 85c] Sha, L., Lehoczky, J. P. and Jensen E. D.
Modular Concurrency Control and Failure Recovery --- Part I: Concurrency Control.
Submitted for publication, 1985.

[Silberschatz 80] Silberschatz, A., and Z. Kedem.
Consistency in Hierarchical Database Systems.
JACM 27:1, 1980.

[Stankovic 83] Stankovic, J. A.
Bayesian Decision Theory and Its Application to Decentralized Control of Job Scheduling.
February, 1983.
Internal documentation, University of Massachusetts, Department of Electrical and Computer Engineering.

[Stefik 82] Stefik, M.; Aikins, J.; Balzer, R.; Benoit, J.; Birnbaum, L.; Hayes-Roth, F.; Sacerdoti, E.
The Organization of Expert Systems, A Tutorial.
Artificial Intelligence 18:135-173, 1982.

[Svobodova 84] Svobodova, L.
Resilient Distributed Computing.
IEEE Transaction on Software Engineering, May, 1984.

[Tokuda 85] Tokuda, H., Clark, R. K. and Locke, C. D.
Archons Operating System (ArchOS) --- Client Interface, Part II.
Technical Report, Department of Computer Science, Carnegie-Mellon University, 1985.

[Weihl 84] Weihl, W. E.
Specification and Implementation of Atomic Data Types.
PhD thesis, Massachusetts Institute of Technology, 1984.

[Wulf 81] Wulf, W. A.; Levin, R.; Harbison, S. P.
HYDRA/C.mmp: An Experimental Computer System.
McGraw-Hill, Inc., 1981.

[Zadeh 79] Zadeh, L. A.
A Theory of Approximate Reasoning.
Machine Intelligence 9.
Wiley, New York, 1979.

4.3 Best Effort Scheduling Model

A Best-Effort Scheduling Model for Real-Time Operating Systems

E. Douglas Jensen, C. Douglass Locke, Hideyuki Tokuda

Computer Science Department
Carnegie-Mellon University, Pittsburgh, PA 15213

This work was supported in part by the USAF Rome Air Development Center under contract number F30602-81-C-0297, the U.S. Naval Ocean Systems Center under contract number N66001-81-C-0484, the U.S. Navy Office of Naval Research under contract number N51160-84-C-0484, and the IBM Corporation, Federal Systems Division. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of RADC, NOSC, ONR, IBM, or the U.S. Government.

Abstract

Process scheduling in real-time systems has almost invariably used one or more of three algorithms: fixed priority, FIFO, or round robin. The reasons for these choices are simplicity and speed in the operating system, but the cost to the system in terms of reliability and maintainability have not generally been assessed.

This paper originates from the notion that the primary distinguishing characteristic of a real-time system is the concept that completion of a process or a set of processes has a value to the system which can be expressed as a function of time. This notion is described in terms of a time-driven scheduling model for real-time operating systems and provides a tool for measuring the effectiveness of most of the currently used process schedulers in real-time systems. Applying this model, we have constructed a multiprocessor real-time system simulator with which we measure a number of well-known scheduling algorithms such as Shortest Process Time (SPT), Deadline, Shortest Slack Time, FIFO, and a fixed priority scheduler, with respect to the resulting total system values. This approach to measuring the process scheduling effectiveness is a first step in our longer term effort to produce a scheduler which will explicitly schedule real-time processes in such a way that their execution times maximize their collective value to the system, either in a shared memory multiprocessing environment or in multiple nodes of a distributed processing environment.

1. Introduction

Process scheduling in a computer operating system is merely an instance of an extensively studied problem from Operations Research (OR), in which it takes the form of producing a sequence of jobs which must utilize a common resource (e.g., a lathe in a machine shop) such that some metric (e.g., number of parts made in a day) is optimized (maximized or minimized). In the real-time operating system as well, the choice of process scheduling algorithms to allocate cpu resources can have a very important impact on the system performance as well as on the reliability and maintainability of the system.

This paper is written from the context of the Archons project, currently under way at Carnegie-Mellon University. Archons is undertaking to create new resource management paradigms which are as intrinsically decentralized as possible, then applying them as the foundation of an experimental decentralized real-time operating system which will then be used in the design and construction of a "decentralized computer"[1, 2, 3]. The process scheduler for this system constitutes one of the critical research interests of the Archons project, which will be expected to manage time resources to ensure that the most important processes meet their time constraints, even in the event of an overload

condition[4].

In the remainder of this section, we discuss real-time systems and the current state of real-time process scheduling, while in Section 2 we define our time-driven scheduling computational model. In Section 3 we describe our experimental simulation results, summarizing our results in Section 5.

1.1. The Scheduling Problem

Scheduling a set of processes consists of sequencing them on one or more processors such that the utilization of resources optimizes some scheduling criterion. Criteria which have historically been used to generate process schedules include maximizing process flow (i.e., minimizing the elapsed time for the entire sequence), or minimizing the maximum lateness (lateness is defined to be the difference between the time a process is completed and its deadline). It has long been known[5] that there are simple algorithms which will optimize certain such criteria under certain conditions, but algorithms optimizing most of the interesting scheduling criteria are known to be NP-complete[6], indicating that there is no known efficient algorithm which can produce an optimum sequence. Clearly, the choice of metric is crucial to the generation of a processing sequence which will meet the goals of the system for which the schedule is being prepared.

There are a number of significant differences between scheduling as it is practiced in most applications of OR and process scheduling in an operating system. In OR, scheduling is traditionally performed statically, off-line, while in an operating system, the information from which scheduling decisions must be made is dynamic, suggesting that the best scheduling decisions should be made dynamically. In addition, classic OR scheduling problems assume (for the sake of computational simplicity, not particularly for realism) that all the jobs to be scheduled and their processing time requirements, are available at the beginning of the sequence ($t = 0$), and that new jobs will not suddenly appear during processing. If a change in any data involved in the scheduling decision (such as the arrival of a new job) should occur, the previously computed sequence is invalidated, and scheduling must be started over if optimality is to be maintained. On the other hand, in the real-time operating system environment, processes frequently arrive and depart (i.e., are completed or terminated) at irregular intervals and have stochastic processing times.

Real-time processes can be partitioned into two categories: periodic and aperiodic. Periodic processes arrive at regular intervals, while aperiodic processes arrive irregularly, but even periodic processes can be terminated or undergo changes in period due to application dependencies, resulting in a constantly changing set of schedulable processes.

1.2. Real-Time Systems

Although many real-time systems have been constructed, the fundamental differences between a real-time system and other computer systems have not always been clearly understood. It is our thesis that the most important difference between the real-time operating system and other computer systems is that in a real-time system, the completion of a process *has a value to the system which varies with time*[7]. Although the time to complete a process is of some importance in all computer systems, in a real-time system the response time is viewed as a crucial part of the *correctness of the application software*; a computation which is late is very frequently no better, or perhaps even worse, than one producing an incorrect result.

This time value is usually described in terms of deadlines by which computations must be completed, and the deadlines are generally traceable to the physical environment with which the system must interact; for example, a reaction temperature measurement must be completed in time to apply a correction in a manufacturing system. Stated another way, the value to the system for completing a process with a deadline is some high constant value which abruptly drops to zero after the deadline.

Real-time systems have been divided into two classes: *hard* real-time systems and *soft* real-time systems[8]. Hard real-time systems are those whose deadlines must absolutely be met or the system will be considered to have failed (and whose failure might be catastrophic), while soft real-time systems allow for some deadlines, at least occasionally, to be missed with only a degradation in performance but not a complete failure.

Many existing real-time systems are being used in a number of such diverse environments as space or airborne platform management, factory process control, and robotics. These involve several levels of real-time, characterized by the tightness of their deadlines, and range from closed-loop sensor/actuator systems (frequently implemented on microprocessors dedicated to a small number of devices) to supervisory systems involving human interaction to manage a complex command and control environment.

1.3. Scheduling in Existing Real-Time Systems

In observing a number of existing real-time systems, we note the characteristics found in the operating systems used managing their manage resources in support of their real-time operation (such operating systems are usually called *executives* since a full operating system is not usually implemented in a real-time system). Two primary characteristics of these operating systems can be described:

- These operating systems are kept simple, with minimal overhead, but also with minimal

function. Virtual storage is almost never provided; file systems are usually either extremely limited or non-existent. I/O support is kept to an absolute minimum. Scheduling is almost always provided by some combination of FIFO (for message handling), fixed priority ordering, or round-robin, with the choice made by the operating system designer.

- Simple support for management of a hardware real-time clock is provided, with facilities for periodic process scheduling based on the clock, and timed delay primitives.

Conspicuously missing from these systems at the operating system level is any support for explicitly managing user-defined deadlines, even though meeting such deadlines is the primary characteristic of the real-time application requirements. Instead, these systems are designed to meet their deadlines by attempting to ensure that the available resources exceed the expected worst-case user requirements, and their implementation is followed by an extensive testing period in an attempt to verify that these requirements can be met under these expected loads.

In fixed priority scheduler systems, deadline management is attempted by assigning a high fixed priority to processes with "important" deadlines, disregarding the resulting impact to less "important" deadlines. During the testing period, these priorities are (usually manually) adjusted until the system implementer is convinced that the system "works". This approach can work only for relatively simple systems, since the fixed priorities do not reflect any time-varying value of the computations with respect to the problem being solved, nor do they reflect the fact that there are many schedulable sets of process deadlines which cannot be met with fixed priorities. In addition, implementers of such systems find that it is extremely difficult to determine reasonable priorities, since, typically, each individual subsystem implementer feels that his or her module is of high importance to the system. This problem is usually "solved" by deferring final priority determination to the system test phase of implementation, so the resulting performance problems remain hidden until it is too late to consider the most effective design solutions. This approach results not only in real-time systems with frequently marginal performance, but also in extremely fragile systems in the presence of changing requirements.

A scheduling algorithm which is seldom used in practice has some interesting properties, but also presents a serious pitfall. A deadline scheduler (i.e., a scheduler which schedules the process with the closest deadline first[9]) solves the problem of missing otherwise schedulable deadlines due to the imposition of fixed priorities, but leaves other problems, most notably transient overloads. Transient overloads occur not only as a result of application and operating system design decisions, but also as a consequence of normal system activity, including interrupt processing, DMA cycle stealing, or blocking on semaphores[10]. The deadline scheduler provides no reasonable control over the choice

of which deadlines must be delayed in an overload, leading to unpredictable failures and resulting in an impact on reliability and maintainability.

The value of a real process completion is well modeled by a step function only in those few cases in which in which there is no longer any value in completing the process after its deadline, such as computing a control parameter for a manufacturing step which has already been completed. In many actual systems, the value of completing a computation after its deadline may rapidly decay but remain positive (e.g., being late on an aircraft navigation update may result in a loss of positional accuracy, while missing it altogether would merely exacerbate the loss unless it became so late that it impacted making the next update), or completing a computation before its deadline may be more or less desirable than completing it exactly at its deadline (e.g., a satellite orbital insertion burn computation, which must occur within a small time "window" around an optimal time). As we show experimentally, process scheduling in the presence of a (possibly transient) overload shows particularly well the difficulty in producing a reasonable schedule with respect to different process' value functions. Determining an appropriate value for a given process consists not only of simply assigning a priority, but rather of defining the system value of completing it at any time. This value is normally defined by the physical application environment which the system must serve.

1.4. Evaluating a Real-Time Scheduler

Given that the primary characteristic of a real-time system is that correctness is determined not only by *what* is done, but *when* it is done, we propose to use a representation of a process completion value to measure the algorithms commonly used or considered for use in a real-time system. In particular, we will simulate a real-time system in which each process has a value expressed as a function of time which defines the value to the system of completing that process at any time, and we will "reward" the system with the value determined by that function when the process terminates. A set of processes which arrive at stochastically determined intervals (regular intervals for periodic processes and Poisson arrivals for aperiodic processes) and compete for a processing resource in a symmetric multiprocessor will then be executed for a given period of time. The sum of the resulting process values for all processes requested during the execution period will then provide our metric for determining the performance of each scheduling algorithm under several conditions of loads, and with several different value functions. This metric measures the long-term (relative to the individual process computation times) performance of the system with respect to its support of the application-defined value of meeting time constraints, a direct measure of the application real-time support provided.

2. Time-driven Scheduling Model

We define a computational model consisting of a set of preemptible processes P resident in a computer with a single shared memory and one or more processing elements. Each process p_i has a request time R_i, an estimated computation interval C_i, and a value function $V_i(t)$, where t is a time for which the value is to be determined[7]. Figure 2-1 illustrates these process attributes for a hypothetical process whose value function is linearly decreasing prior to a critical time D_i with an exponential value decay following D_i. The process illustrated depicts a process which has been dispatched shortly after its request time and which has compieted prior to its critical time without being preempted.

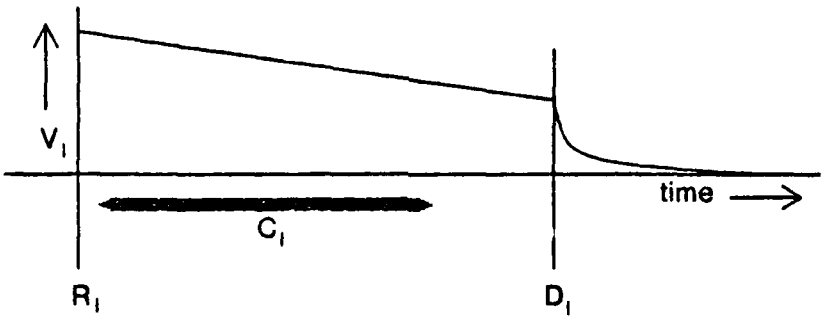


Figure 2-1: Process Model Attributes for Process i

$V_i(t)$ defines the value to the system for completing P_i at time t. The nature of V_i is determined by the range of scheduling policies supported by the operating system, particularly with respect to the handling of a processor overload in which some critical times cannot be met. The value function could come from either or both of two sources: the process implementer who understands the requirements of the internal environment as it affects that process, and the system architect who is aware of the relationship between that process and the overall system, including its external environment. For the purposes of this paper, we assume that all sources of value information are reflected in the value functions used.

We note that the existence and importance of a deadline for p_i is dependent on nature of its value function. The value function can be said to define a critical time (also called a deadline if the function is a step function) only if it has a discontinuity in the function or its first or second derivative. Functions of this type are characterized by a relatively rapid change in value with respect to time, with the deadline case only an extreme example. For example, the process illustrated in Figure 2-1, has a critical time defined by the discontinuity in its first derivative at D_i. For convenience in the discussion of the simulator, we will actually refer to a critical time as the time axis origin relative to which the value function for a process will be computed, even if the function has no discontinuity as described above.

The use of value functions as described in this model allows us to describe both hard and soft real-time environments, and, in particular, allows us to evaluate systems which mix deadlines of both types in a single system. Since the value functions used in this model may or may not explicitly define either a critical time or a deadline, and the critical time may not actually constitute a deadline, in the remainder of this paper we will avoid the use of the word "deadline", which implies the step value function. Hence, we will refer to the time of the value function discontinuity, if any, as its critical time.

The request time R_i has been defined in our simulation (see Section 3) as an arbitrary time at which P_i has been requested to be executed. The significance to the scheduler as we have defined it is that the process is not schedulable prior to P_i. Following P_i, it remains schedulable until one of the following conditions has occurred:

- It has completed,
- Its value function has become zero or negative.

We note that the value function may be negative at R_i, not rising above zero until a later time (see Section 3 for an example of such a function).

Thus, as viewed by the process scheduler, the request time R_i for a process may be either a future or a past time. If R_i is a future time, the process is not currently schedulable, but its attributes may be considered in the current computations of load from which current scheduling decisions are made.

The computation time C_i is a random variable representing the expected execution time to process P_i (estimated time remaining if P_i has already begun processing), not including system overhead or preemptions. The source of this value and its distribution for a real operating system would be either the programmer or an actual measurement by the system itself (see Section 4), but here we will simply assume that it is available to the scheduler. Clearly, the value function can be seen as a definition of application scheduling policy, and although in a real system not every policy would be subsumed in a single set of value functions, in this paper, we will assume that the value function completely defines the policies to be implemented by the scheduler. The relationship between our value functions and the global scheduling policy is a subject of continuing research in this effort.

Given the potentially diverse set of value functions which could be produced in a given system, we will show that none of the normally used scheduling algorithms can consistently produce a good schedule as the processing resources approach saturation (see Section 3). As an example, see Figure 2-2 for an example of four processes in a single processor with value functions, for which the best choice of an execution sequence is non-trivial. The figure shows the four value functions, and a

potential scheduling sequence such that each completes with a high value. We will describe later the significance of such value functions in the discussion of the experiments conducted.

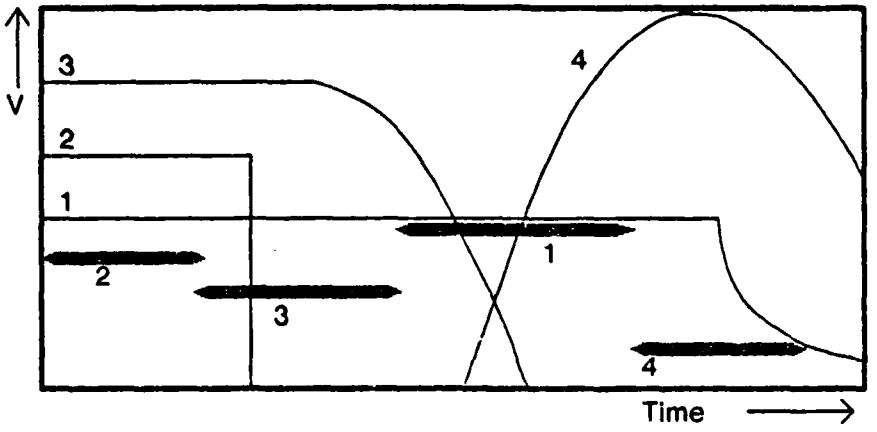


Figure 2-2: Four "Typical" Processes with Value Functions

This model encompasses both periodic and aperiodic processes in that any single execution of a periodic process can be described as shown above in exactly the same way as for a non-periodic process. The only difference is that the future request times for a periodic process are known in advance. We allow overlapping executions by allowing multiple instances of a process to be simultaneously schedulable (we assume that processes are reentrant).

It should be noted that precedence and consistency (e.g., processes involved in mutual exclusion) relations are not addressed in this model at this time, since it has been assumed that the scheduling algorithm will consider only processes which are ready to be executed and for which a request time has been determined. If one process is dependent upon completion of another, the second process' request time will not have been determined until the first process has been completed.

3. Simulation Results

3.1. Real-Time Scheduling Simulator

To provide an environment in which various scheduling algorithms can be evaluated with respect to their performance in generating a high total system value, a simulator has been constructed which provides an operating system environment in which each of the scheduling algorithms can be executed and evaluated.

The value functions to be used by the simulator are not competely general forms, but rather are limited to a specific form possessing characteristics making the expressiveness of the value very

flexible while allowing the analysis of the resulting functions to be as tractable as possible. We define the value function in two parts; the value prior to the critical time and the value after the critical time, providing for a discontinuity at the critical time if desired by the application designer. For each of these two parts, five constants are used to define the value function relative to the critical time using the expression:

$$V_i(t) = K_1 + K_2 t - K_3 t^2 + K_4 e^{-K_5 t}$$

The simulator first uses a statistical model of a "typical" real-time processor load to generate a set of processes, including both periodic and aperiodic processes. The statistical model defines a real-time workload based on experience with several actual real-time systems with which this author is familiar. The processor load generated can be varied to simulate both heavily loaded systems and relatively lightly loaded systems. The nature of this statistical load, including the process characteristics, the simulated hardware characteristics, and the value functions, can be easily modified during the experimentation allowing us to evaluate a number of systems with different characteristics. It is also possible to impose an additional "spike" load in which the aperiodic arrival rate dramatically increases at certain periods during the simulation, allowing us to determine scheduler performance during transient overloads. Our test process sequences will emphasize the overloaded condition, since that is the condition of primary interest.

Once a set of processes has been constructed, the simulator "executes" the processes, applying a statistical model to generate the actual request times and computation times for each process. Execution consists of iteratively calling the selected scheduler for its decisions, "executing" the selected process, updating the clock to the next important time (e.g., time of arrival of a new process), then asking the scheduler for its next decision.

3.1.1. A Set of Classical Algorithms

The simulator has the ability to make multiple runs sequentially using an identical set of processes with identical values and request times, using various "classical" algorithms. These "classical" algorithms include:

1. **SPT**. At each decision point, the process with the shortest estimated completion time is executed.

2. **Deadline**. At each decision point, the process with the earliest critical time (interpreted by this algorithm as a deadline) is executed.

3. **Slack**. At each decision point, the process with the smallest estimated slack time (elapsed time to the critical time minus its estimated completion time) is executed.

4. **FIFO.** At each decision point, the process which has been in the request set longest is executed.

5. **Random.** At each decision point, a process is chosen (with uniform probability) from the request set and executed.

6. **RandPRTY.** At each decision point, the process with the highest fixed priority is executed. In the runs reported, this fixed priority was chosen to be the same as the highest value reached by the corresponding value function.

The simulator keeps a large number of statistics on each run, including such parameters as the number of tardy processes, the maximum lateness, the average lateness, and the total value accumulated. This data is saved and reported for each real-time run made in a simulation sequence, and is reported in a statistical summary at the end of the simulation, resulting in the data provided below.

3.1.2. A Pair of Interesting New Algorithms

In addition to these algorithms, two experimental algorithms have been implemented, and have produced some initially promising results which we include in the experiments demonstrated here. An important part of our research effort consists of the further development of algorithms such as these, and the concepts behind them, as they apply to a real-time system.

In these initial algorithms, we take advantage of three observed value function and scheduling characteristics:

1. Given a set of processes (ignoring deadlines) with known values for completing them, it can be shown that a schedule in which the process with the highest value density (V/C, in which V is its value and C is its processing time) is processed first (i.e., a Value Density Schedule) *will produce a total value at every point in time at least as high as any other* schedule.

2. Given a set of processes with deadlines *which can all be met* (based on the sequence of the deadlines and the computation times of the processes), it can be shown that a schedule in which the process with the earliest deadline is scheduled first (i.e., a Deadline schedule) will always result in meeting all deadlines.

3. Most value functions of interest (at least among those investigated at this time) have their highest value occurring immediately prior to the critical time.

Some of the implications of these observations are:

- If no overload occurs, the deadline schedule will have all deadlines met, and no higher total value will be possible.

- If an overload occurs, and some processes must miss their deadlines, the Value Density Schedule would produce a high value, but might miss some deadlines which could

otherwise have been met.

Our first algorithm (called the BEValue1 algorithm in the experimentation description, section 3.2) exclusively uses observation 1 above, and is therefore a simple greedy algorithm scheduling first the process with the highest expected value density. This algorithm actually performs reasonably well in many cases in which the value function is a step function or rapidly decreasing following the critical time, in spite of the fact that it makes no explicit use of the critical time itself. The critical time does, of course, enter the algorithm through the expected value computation, which uses the value function and the assumed computation time distribution to compute the expected value. This algorithm fails most notably in step function situations in which no overload is present and a number of processes with close deadlines are in the request set. In this case, the processes with high value density will be run, quite possibly preventing other processes from meeting their deadlines even though all deadlines could have been met.

Our second algorithm is a modification of the simple deadline algorithm, attempting to remove its most important failing; in an overload the deadline scheduler will give priority to processes whose deadlines cannot possibly be met, delaying other processes which could still meet their deadlines. Therefore, after computing the probability of an overload, we choose processes with low value density as candidates for being removed from an overloaded deadline schedule until a deadline schedule is produced which has an acceptably low probability of producing an overload.

This algorithm (called the BEValue2 algorithm in the experimentation description, section 3.2), starts with a deadline-ordered sequence of the available processes, which is then sequentially checked for its probability of overload. At any point in the sequence in which the overload probability passes a preset threshold, the process prior to the overload with the lowest value density will be removed from the sequence, repeating until the overload probability is acceptable. This algorithm seems generally to outperform BEValue1, since it always meets deadlines as long as no overload occurs, and transitions gradually into the same performance as BEValue1 as an overload condition worsens. In this algorithm, modifications to the overload probability threshold can significantly affect its overload performance; at extremes, a threshold of 100% results in a pure deadline schedule, while a probability threshold of 0% results in a BEValue1 schedule. For the experiments described here, a threshold of 40% is used. Optimizing this threshold as well and the other critical values used in the heuristics from which BEValue2 is constructed are goals of our continuing research in this area.

3.2. Experiments Executed

To test our hypothesis that the efficacy of a scheduling algorithm in a real-time system can be measured with respect to the key distinguishing characteristic of such systems, namely that completing a process has a time-varying value to the system, we have generated a set of simulation experiments using several well-known scheduling algorithms. As with any such set of measurements, this set is necessarily incomplete, but is intended to illustrate some of the characteristics which would be observable with systems under several potential conditions.

Each experiment described here was run on a simulated, symmetric, shared-memory multiprocessor, with the number of processing elements varying from one to four (thus testing the schedulers under four load levels). All processes were considered to be fully preemptible and the effects of context switching and scheduling overhead were neglected. Thirty-six processes were selected, each with an individually determined stochastic load assumed to be normally distributed and characterized by its mean and standard deviation. The mean loads of these processes were themselves approximately normally distributed across the 36 processes, with a mean of 500 ms. and a standard deviation of 300 ms., with the limitation that the minimum process load was 1 ms. Figure 3-1 lists the actual set of processes, showing for each its process ID, its period (if it is periodic), its load characteristics, and its critical time.

As each simulation progressed, each process from this list was requested at either the periodic rate, if it was a periodic process, or using an exponential interarrival time with a mean of 10 seconds. In addition to these requests, additional aperiodic processes arrived as a "spike" load with a mean interarrival time of 1 second, exponentially distributed, every 10 seconds, with the "spike" lasting 200 ms. This simulated set of requests continued for 60 seconds of elapsed time, at the end of which the requests ended and the unfinished processes in the request set, if any, were completed by the multiprocessor, terminating the simulation when the queue was empty. At the completion of each process, its value was computed and the values from all completed processes are summed, producing a final total value over the 60 second request interval. No value was accrued for processes not competed (i.e., aborted). As the list in Figure 3-1 shows, four of the processes were periodic; the total system load from these four processes represented about 42% of the total process load. This simulation resulted in the arrival of about 305 processes, including both periodic and aperiodic processes.

Four value functions, illustrated in Figure 3-2, were used in separate executions to compute the total value generated by each of the scheduling algorithms under differing conditions of perceived system value.

ID	Period	Mean Load	Load σ	Critical Time
1		0.474	0.058	0.683
2		0.045	0.009	0.128
3		0.580	0.159	1.746
4		0.516	0.141	1.267
5		0.492	0.132	1.069
6		0.133	0.034	0.193
7		0.627	0.129	1.402
8		0.251	0.044	0.444
9		0.859	0.271	1.894
10		0.520	0.181	0.762
11		0.574	0.080	1.405
12		0.217	0.027	0.674
13		0.920	0.166	2.752
14		0.737	0.135	1.749
15	1.731	0.247	0.069	0.618
16		0.515	0.067	1.159
17		0.748	0.345	2.160
18		0.696	0.248	1.390
19	5.762	0.782	0.229	1.260
20		0.555	0.088	1.422
21		0.805	0.197	2.490
22		0.051	0.003	0.148
23		0.684	0.166	1.583
24	11.119	0.827	0.215	1.932
25		0.408	0.133	1.254
26		0.001	0.000	0.002
27		0.894	0.185	2.126
28		0.754	0.166	2.119
29		0.387	0.099	1.166
30		0.658	0.141	1.606
31		0.249	0.103	0.562
32		0.459	0.112	1.409
33		0.650	0.100	1.788
34		0.498	0.152	0.714
35	3.663	0.245	0.045	0.618
36		0.130	0.041	0.322

Figure 3-1: Simulation Experiment Process List

The first (V1), showing a constant value prior to the critical time followed by an exponential decay after the deadline, represents a case such as a vehicle navigation problem in which a position update must be completed by a given time each cycle to provide a specified precision, but if late, the resulting error can be minimized by making up the computation, assuming the next navigation cycle is not overrun. As the next cycle approaches, the value of running the previous one becomes negligibly small. The second (V2) represents a hard deadline process, in which there is value to the system for completing it only if it precedes a fixed deadline, such as for certain sensor inputs, in which the value to be read is no longer present if a fixed time interval is exceeded. The third value function (V3) is

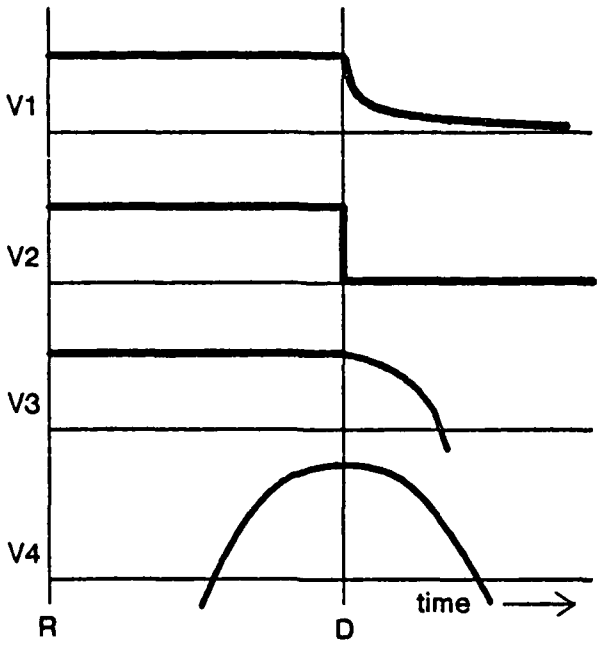


Figure 3-2: Example Value Functions

constant prior to the deadline, but drops off parabolically after the deadline, representing a process similar to the hard deadline above, but with a softer deadline as long as the process is not excessively late (in this experiment, the value function drops past zero at about 516 ms. following the critical time). The fourth function (V4) is parabolic both before and after the critical time, representing a process whose completion should be delayed until after some arbitrary time, such as a satellite launch, in which the desired orbit cannot be reached prior to a given time or after a later time (in this experiment, these functions pass zero approximately 516 ms. before and after the critical time).

In an actual system, it would be expected that each process in a given application would have a distinct value function, but we have used identically shaped value functions for every process in the run to isolate the effects of multiple value function shapes on scheduling quality. For each of the four value functions chosen, four experiments were conducted, with 1, 2, 3, and 4 processing elements each, representing average process loads of 246%, 123%, 82%, and 61%, respectively, and each simulation was repeated 15 times, averaging the computed parameters. For each algorithm in each experiment, several parameters were computed:

- *Total Value(V1, V2, V3, V4)* -- The sum of each of the four value functions for each process computed at its completion time, averaged over the 15 runs.

- *Mean Preemptions(MP)* -- The number of times a running process is preempted by a more critical process.

- *Mean Lateness(ML)** -- The difference between the actual process completion time and its critical time, averaged over the 15 runs. This value is negative if the process completes prior to its critical time, and positive after the critical time.

- *Maximum Lateness(MaxL)* -- The largest lateness value of the entire 60 second run, averaged over the 15 runs for which each simulation was repeated.

- *Mean Tardiness(MT)* -- The average of the non-negative latenesses over the 15 runs.

- *Number of Tardy Processes(NTP)* -- The total number of processes completing after their critical times, averaged over the 15 runs.

and the results of this measurement for the standard algorithms are graphically represented in Figure 3-3, while the more specialized BEValue1 and BEValue2 algorithms are represented in figure 3-4. In these figures, each circle represents a particular scheduling algorithm executing under a particular load. Within each circle, each line represents one of the measurements described above, with a longer line indicating that the corresponding measurement was one of the best for that load value, while a short line indicates that the corresponding measurement was relatively poor. Although these measures are all reported for each set of runs, the most important measures from our perspective are the four value function results, V1, V2, V3, and V4.

Processes whose value has passed below zero are aborted, and the resulting system value is zero, reflecting the fact that they produce no contribution to the system performance. This abortion occurs for all algorithms tested.

3.3. Experimental Results from Standard Scheduling Algorithms

There are a few observations which can be made from these measurements across all runs. It should be noted that in no case did the FIFO scheduler or the random scheduler generate the best schedule, although they produced acceptable schedules when the system was sufficiently underloaded; it is clear that neither of these would be the algorithm of choice unless we were guaranteed that an overload condition could not occur. Both of these algorithms are used extensively in certain types of actual systems; FIFO is used for many message passing schedulers, and random scheduling is effectively used for some contention bus communications links (e.g., Ethernet). Similarly, the fixed priority scheduler performed inconsistently, particularly as the overload condition became more pronounced. Thus, of these algorithms, the choice would be between the SPT and Deadline scheduling algorithms, neither of which are widely used in existing real-time systems.

---

*Value functions resulting in aborting processes (i.e., all the value functions used in the experiments reported here except the exponential decay function [V1]) produce meaningless values for the lateness and tardiness, so the reported measurements for these are included only for runs which result in no abortions.

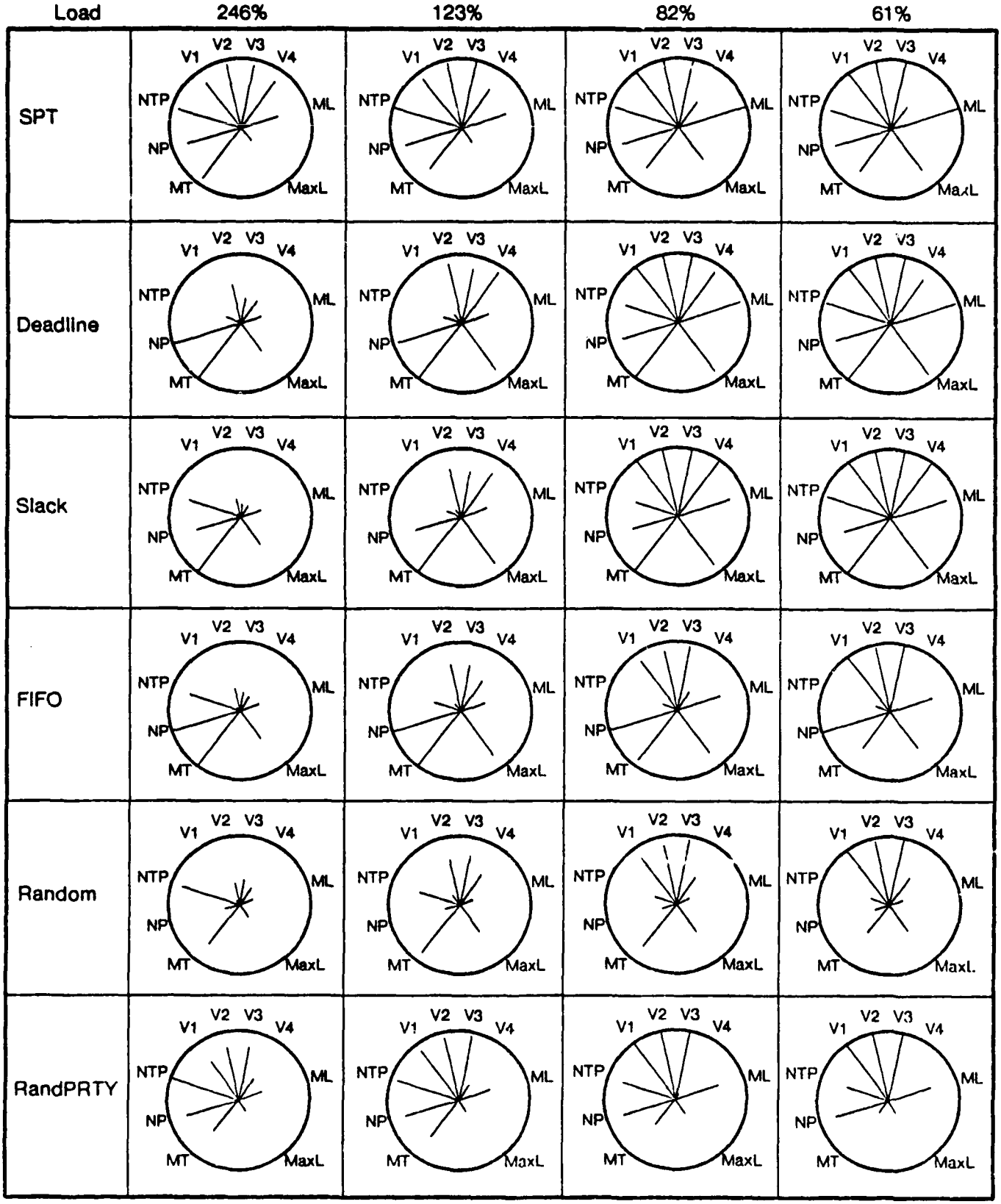


Figure 3-3: Standard Algorithm Experiment Results

With respect to the observed value variations among the different processor loads, the first observation is that, except for the fourth value function (V4) runs (parabolic value functions both prior to and following the critical time), the scheduling function makes little or no difference when the system is lightly loaded. Although in the underloaded case the deadline scheduler generally shows the best performance in most of the relevant measured parameters, the difference between the best and worst among the six algorithms with respect to total value is within about 5% on value functions V1, V2, and V3. As expected, we note that the insensitivity to the algorithm is most pronounced at the average load of 61%, and less so at 82%, where the preference for the deadline and slack time schedulers are more pronounced (and the difference between the best and worst total values) is about 12%. This result confirms our intuition that the scheduling algorithm used for a suitably underloaded system is not usually an extremely critical factor. The case with value function (V4) is radically different in that the value functions indicated that the processes should have had their executions delayed to achieve acceptable values, but none of these algorithms took this effect into account. Such an effect is typically overcome in existing real-time systems by the application processes themselves introducing delays in their processing, and relying on priority to force the delayed process to execute properly in its value function "window".

The picture becomes more complex as the load increases to an overload condition. At 123% overload, the deadline scheduler is beginning to have considerable difficulty with V1, V2, and V3, and the SPT scheduler seems to be performing somewhat better. Even the fixed priority scheduler is doing better than the deadline scheduler. This effect, while perhaps not intuitive, can be explained by noting that the SPT algorithm is designed to complete the largest possible number of processes in a given time interval, increasing the likelihood that the short processes will complete prior to their deadlines and resulting in a greater number of processes doing so, therefore producing a higher total value. The deadline scheduler, while ensuring that deadlines will be met if there is sufficient processing capacity, fails to satisfactorily handle an overload due to its propensity for giving priority to processes which are about to miss their deadlines over those whose deadlines are still to come.

3.4. Experimental Results from Value Function Algorithms

Of the two experimental value function algorithms described above, it is clear that the BEValue2 scheduler is a clear winner as shown in Figure 3-4, plotted with identical scales as Figure 3-3, using the identical value functions and loads. It outperforms all other tested algorithms in all the overload runs by significant margins, and shows a consistency of performance which none of the other algorithms can approach. Like the others tested, it had some difficulties with the V4 function, since it, too, takes little account of the fact that the processes needed to be delayed to achieve a high value,

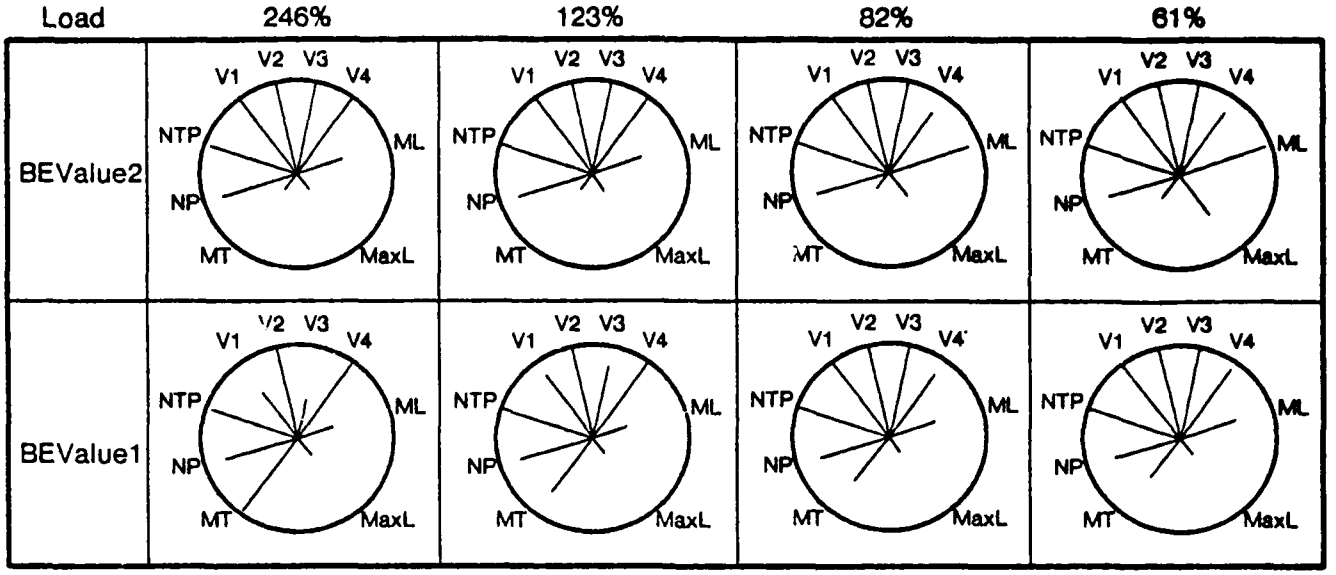


Figure 3-4: Value Function Experiment Results

and this is one of the critical areas in which further research will be performed. These results should be viewed as preliminary at best, but it is certainly clear that it has some important characteristics which would be needed in a value function scheduler.

The BEValue1 scheduler performance was, like many of the other algorithms, much more variable, depending on the value function and load. Because it is a "greedy" algorithm, preferring to pick up value early rather than wait to get a higher value, it misses many opportunities to meet time constraints.

3.5. Experimentation Summary

It is clear that the shape of the value function and the processing load level has a great effect on which algorithm is better for the total system performance when the system is overloaded. This experimentation seems to indicate that, with improvements, a scheduler which overtly uses the application-defined system value to handle scheduling should, at least in principle, provide a consistently better schedule. In fact, such improved performance, particularly in overload conditions, does result from our two experimental scheduling algorithms, although our analysis on them is by no means complete at this time.

4. Real-Time Operating Systems Interface

There are many potential approaches to utilizing a time-driven scheduling model in a real-time operating system. Although traditional real-time operating systems provide a simple set of time control primitives (e.g., GetTime, Schedule, Timeout), no system primitives are available to explicitly express *request* R_i, *critical* D_i, *estimated computation times* C_i, or *a value function* $V_i(t)$ for each process. Furthermore, a real-time operating system should be able to support a primitive to modify the value function for a process or a set of processes during run time, as well as primitives to specify a system-wide scheduling policy. In this way, an application designer can set up and modify a suitable scheduling policy under various system conditions.

For the purpose of describing these operating system primitives, we assume that primitives to create and kill processes already exist. We express each primitive as if it were implemented as a procedure in a high-order language such as C or Pascal.

4.1. Time Control Primitives

The arguments to these operating system primitives communicate the information needed to implement this model, but an important issue is the structure of the information to be passed to the operating system. In the following primitives, the structure of the passed information has been designed to encourage internal consistency. Using a single primitive to define each parameter would be extremely flexible, but a user might inadvertantly set mutually inconsistent parameters. Thus, the system should provide a simple way to define a consistent set of parameters.

The *Delay* primitive

```
DateTime = Delay(DelayTime, CriticalTime,
                 CompTime, SetFlag)
```

blocks the requesting process a specified length of time (e.g., in microseconds), then returns the current date and time when the delay is completed and execution has resumed. This primitive would also be used to specify the critical time D_i (see Section 2) and an estimate of the amount of processing time that will be required to reach the critical time** C_i, as well as to mark the arrival of the process at the critical event for which the critical time was established (while optionally defining the next critical time) by setting "SetFlag" value. If a value of "DelayTime" is not positive, the system would not block the process, and similarly the estimated execution time and critical time would not be changed if its value were not positive.

**A system may also estimate this processing time based on observations of earlier executions.

For instance, a simple command to alter the critical time can be defined by using the *Delay* primitive:

Delay(0, CriticalTime, 0, TRUE)

4.2. Periodic Processes

While the *Delay* primitive provides time dependent processing control at runtime, periodic repetitive processing can be specified at process creation time.

One way to describe a periodic process is by using optional arguments in a *CreateProcess* primitive:

pid = CreateProcess(ProcessName, InitialMsg,
PERIODIC, DelayTime, Period)

The "CreateProcess" primitive would create a new instance of a process at a specified node. "InitialMsg" indicates an initial message to be immediately sent to the new process and "PERIODIC" specifies that the new process will be periodic. "DelayTime" sets the first request time for the new process and "Period" is the indicated period.

4.3. Scheduling Policies

For a practical real-time operating system, it is necessary to provide a mechanism to express the application-defined scheduling policies needed to implement our scheduling model. The system should be able to modify these policies at runtime in order to take advantage of the flexibility of the model. The value function of the time-driven model could be provided as an executable function by the user, or the user could express various scheduling policies by defining a dynamic set of values K_1, K_2, ..., K_{10} (see Section 3), so that a client need only pass these values to define a new value function $V_i(t)$:

SetValueFunction(K_1, K_2, ..., K_{10}), or

SetValueFunction($V_i()$)

The first primitive is acceptable as long as the application does not require new value function which cannot be expressed by the fixed value function. The second primitive has greater flexibility in terms of passing an arbitrary value function from a client to the system, but, since the actual code of the value function would then be provided by the client, the scheduler would be required to compute its value in user's address space every time it is needed. Thus, this scheme may introduce a significant overhead to the scheduler and, in addition, such flexibility would make it very difficult to develop a future scheduler able to use a value function explicitly to make scheduling decisions.

Our approach is to use the notion of "policy/mechanism" separation[11], to increase the flexibility and reduce scheduling overhead. The basic idea is to create a user-definable system module, called

policy definition module which consists of a *policy body* and a set of *policy attributes*. A policy definition can be placed in a kernel, a system process or a client process as needed to control the system overhead. A *policy definition descriptor* would indicate the location of the policy definition and the information related to the policy body and attribute set.

For instance, a policy definition module could be defined and the policy attribute for a specific policy could be set using the following primitives:

```
SetPolicy(PolicyName, PolicyDefinitionDescriptor)

SetAttribute(PolicyName, AttributeName,
             AttributeValue)
```

The *SetPolicy* primitive would bind a user-defined policy definition module to the operating system. "PolicyName" indicates a symbolic name of policy definition module and "PolicydefinitionDescriptor" points to a data structure which describes the policy definition module's location and its properties. The *SetAttribute* primitive sets the value of a specified attribute for the specific policy module.

For example, the following calls could be used to replace the the *SetValueFunction* primitive above.

```
SetPolicy( SCHEDULE, MyPDD )

SetAttribute( SCHEDULE, "VALUEFUNCTION",
              K1, K2, ..., K10 )
```

This *SetPolicy* primitive would bind a scheduling policy module described by "MyPDD" to the operating system. In the case of the scheduling policy module, we would place the proposed value function at the kernel level. The *SetAttribute* primitive then defines a shape of the value function by giving the values K_1, K_2, ..., K_{10}. We believe that such a proposed value function is rich enough to express a powerful set of scheduling policies and we have reduced the scheduling overhead while increasing scheduling flexibility by not attempting to pass the arbitrary function $V_i(t)$ itself.

5. Summary

Process scheduling is a frequently overlooked determinant of real-time performance. It is our contention that time value of process completion, even though it is the primary characteristic of a real time system, has been neglected in measuring real-time performance. We have illustrated some of the effects of varying the scheduling algorithms, particularly in the presence of an overload condition, relative to several value function characteristics.

This use of value functions to describe the performance of existing scheduling algorithms represents

a first step in our goal of eventually producing a scheduler along the lines of BEValue2 which will explicitly use such value functions to make scheduling decisions[4]. Such a scheduler would attempt to enhance the overall system value by ensuring that processes providing the highest value to the system are favored in the event that some processes must be delayed or aborted due to a (possibly transient) overload. This would provide a level of predictability to overload processing in a real-time system which is not generally available at present, and should therefore allow more effective use of the available resources, lowering system cost.

This paper has presented a preliminary overview into a significant research effort recently initiated within the Archons project. The goals of this work include further understanding the scheduling problem in the presence of user-defined value functions, determining the heuristics necessary to create a tractable value function scheduler, understanding the sensitivity of such a scheduler to the parameters defined by its heuristics, and analyzing the decisions made by those heuristics. Additional measurements with modifications to the scheduling algorithms could be made providing for such capabilities as defined lower bounds for use in process abortion when the value function drops below zero, as well as studies of the effects of varying the heights and shapes of the value functions among the processes. The methods for determining which value function to use for specific types of devices and algorithms need to be studied, the implications of value function scheduling on the specification of scheduling policy, methods of decomposing value functions for a number of processes cooperating to perform a single application function need to be determined, and the use of value functions in the presence of process-to-process dependencies and mutual exclusion must be investigated. Following the initial effort, this research will be extended by the Archons project for process scheduling on a distributed system.

Acknowledgements

Professor Jensen created the initial concepts and development of continuous time-valued scheduling for real-time systems. Mr. Locke, in his Ph.D. thesis research, is further developing the model, defining the required heuristics, and evaluating them through simulation. Dr. Tokuda is responsible for the incorporation of these ideas into a scheduling facility for the Archons project's ArchOS operating system.

The authors are greatly indebted to the many members of the Archons project, especially Professor John Lehoczky and Dr. Lui Sha, who provided considerable assistance in the definition of these concepts, as well to the Archons sponsors.

References

1. Jensen, E. D., *Distributed Control*, Springer-Verlag, 1981, pp. 175-190.

2. Jensen, E. D., "Decentralized Executive Control of Computers", *Proceedings of the Third International Conference on Distributed Computing Systems*, IEEE, October 1982, pp. 31-35.

3. Jensen, E. D., "ArchOS: A Physically Dispersed Operating System", *IEEE Distributed Processing Technical Committee Newsletter*, June 1984.

4. Locke, C. D., "Best-Effort Decision Making for Real-Time Scheduling", Ph.D. Thesis Proposal, Carnegie-Mellon University, Computer Science Department

5. Conway, Maxwell, and Miller, *Theory of Scheduling*, Addison-Wesley, 1967.

6. Garey, M. R.; Johnson, D. S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.

7. Jensen, E. D., Private Communication based on his Honeywell Systems and Research Center technical report on this topic for the U.S. Army Ballistic Missile Defense Advanced Technology Center.

8. Mok, A. K., *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*, PhD dissertation, Massachusetts Institute of Technology, May 1983.

9. Liu, C. L.; Layland, J. W., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of the Association for Computing Machinery*, Vol. 20, No. 1, January 1973, pp. 46-61.

10. Lehoczky, J. P.; Sha, L., publication to appear.

11. Wulf, W. A.; Levin, R.; Harbison, S. P., *HYDRA/C.mmp: An Experimental Computer System*, McGraw-Hill, Inc., 1981.

4.4 Dynamic System Reconfiguration

Dynamic System Reconfiguration

1 Introduction

In this section, we summarize our work on best effort decision making algorithms for dynamic system reconfiguration. The objective of dynamic system reconfiguration is to re-organize a system to adapt to a changing tactical environment or to overcome hardware failures. The study of reconfiguration algorithms is important to distributed tactical systems that must operate in hostile and ever-changing environments.

Conceptually, the three goals of reconfiguration are minimal disturbance, minimal vulnerability and maximal feasibility, and the ability to reconfigure the current task set without shedding load. The goal of minimal disturbance requires the reconfiguration algorithm not to disturb the critical modes in functioning processors, so that the system can carry out its mission without much additional disruption. The goal of maximal feasibility focuses on the restoration of as many disabled modes as possible. The goal of minimal vulnerability is introduced to ensure that as a result of reconfiguration the system is made as fault tolerant to failures as possible, so that the impacts of future system failures are lessened. Conceptually, the notions of minimal disturbance and vulnerability help to eliminate the undesirable regions in the solution space, while the notion of maximal feasiblity seeks to have as large a solution space as possible.

We have successfully developed a system of algorithms that

1. minimize the disturbance to the functioning tasks in the course of reconfiguration,
2. reduce the vulnerability to future system failure,
3. and operate in real-time.

2 Problem Complexity and Algorithm Design Decisions

In a distributed system, the reconfiguration task belongs to the class of *quadratic integer programming* problems. The term "quadratic" refers to the pairwise nature of the communication traffic and the term "integer" refers to the indivisibility of a process. The determination of the *optimal* solution to this type of problem is known to be a special type of NP-complete problem, which, except for very small sizes, "will remain intractable perpetually" [Karp72]. To appreciate the difficulty, we report the work done by Kaku and Thompson [Kaku83]. They used one of the best known algorithms coded in Fortran on a DEC-20. It took on average 31.43 seconds to obtain the optimal solution for an 8 by 8 problem, 220.91 seconds and 1,305.65 seconds for 9 by 9 and 10 by 10 problems, respectively. The rapid exponentially increasing computation time is characteristic of this type of problem. Owing the the very nature of this problem, we had to give up the search for optimal solutions and develop our own hybrid system of fast algorithms for the reconfiguration

problem.[1]

3 Basic Concepts: Disturbance and Vulnerability

When a failure of a processor or other hardware element occurs, there is an immediate interruption of various functioning processes either because the processor in which they were running failed or because the communication paths being used were disrupted. It is important to realize that processes are the fundamental unit of reconfiguration; however, modes are the fundamental unit of disturbance. A system mode consists of a collection of processes. If any of those processes fails (because a processor fails or a communication path is interrupted), the mode is disrupted. Thus, modes are the unit of the system functionality.

One must now reconfigure the system to overcome the failure. This can be done by rerouting messages, relocating processes, shedding load or all of these three. There are three distinct aspects to be considered:

1. The present disturbance incurred by the failure.
2. The additional disturbance caused by reconfiguration.
3. The vulnerability of the subsequent reconfigured system.

We next analyze each of these aspects. Let us begin by quantifying the disturbance.

3.1 Quantification of Disturbance

In a real time command and control environment, a system will require a large number of modes to function properly. Clearly, some modes will be of greater importance than others. We will assume that it is possible to attach an *index of criticality* to each of the modes and that this set of indices is available to the reconfiguration algorithm. This index is a measure of the importance of one mode relative to another fixed reference mode. This index can be used as *scoring device* by which to measure the amount of disturbance caused by an interruption of a set of modes. The scoring of disturbance is a very serious issue. It is most convenient to adopt a *linear scoring* rule. This rule would operate as follows. If a set of modes ceased to function, then the total disturbance is taken to be the sum of the individual indices of criticality. This is a reasonable approach in most cases and is generally very convenient; however, there are instances where it is inadequate and even misleading. For example, suppose that we consider two modes associated with scanning: scanning mode 1 and scanning mode 2. Each may have the same index of criticality. The loss of a single scanning mode is not catastrophic, because there is a second which can fulfill part of the scanning function. If, however, both scanning modes are lost, the impact is very serious, far greater than the sum of the two index

[1] The essential difference between an optimal algorithm and a fast algorithm in this context is that the former must cover the entire solution space, while the latter limits the search to "promising areas" of the solution space and uses simple and fast operations. Fast algorithms are typically greedy and generally give good solutions. However, fast algorithms cannot be expected to find optimal solutions.

scores, because the entire scanning function has been lost. This situation is representative of a *non-linear scoring* problem. That is, the total score is not simply the sum of the individual scores.

An alternative to a linear approach is a *utility function* approach. We must define a function which takes sets of disrupted modes and evaluates them numerically. This function can be as general as the situation warrants and must be constituted by hand for all the possible failure subsets. This is a feasible, but difficult and exacting task.

We propose to adopt a hybrid, intermediate scoring approach. This approach begins with a linear scoring rule but the reconfiguration algorithm will take the non-linearities into account. Furthermore, we do this in a simple and fast algorithmic way. This approach is as follows. We consider each of the modes separately and define an index of criticality for each. Next, we consider the modes in pairs. For each pair, we determine whether the simultaneous failure of each would be of profound seriousness. If so, we create an index of joint criticality and define a pseudo resource of 1 unit for this pair. Both modes are assumed to require a 0.6 units of this pseudo resource, while all other modes require 0 units of this resource. We continue in this fashion creating critical pairs, indices of joint criticality and pseudo resources. One can continue with triplets of joint criticality indices and pseudo resources. In this case, each member of a triplet requires 0.4 units of the pseudo resource, while all other modes require 0 units of it. The result of this approach is that the reconfiguration algorithm will be able to avoid critical joint disturbance and avoid putting jointly critical modes in the same processor which would otherwise create an unnecessarily large future vulnerability.

The creation of pseudo resources addresses the non-linear scoring problem and allows us to keep a linear scoring approach. Notice, however, that each additional pseudo resource brings new constraints into the reconfiguration algorithm and hence reduces reconfiguration potential. Consequently, one should use this approach sparingly.

3.2 Future Vulnerability

When a failure occurs, certain modes will be disrupted and a criticality score can be computed using the linear rule and joint indices. After examining the impact of the failure, the system must be reconfigured, and additional disturbance may occur owing to the reconfiguration. The initial part represents *sunk cost*, the cost from the failure. The reconfiguration part can only add to the cost. Therefore, there is a powerful incentive to incur the smallest possible additional cost in reconfiguration. This is, however, shortsighted. The sunk costs which seem to be unavoidable were actually the result of the previous reconfiguration decision. The reconfiguration from the previous failure left the system in a state which lead to the current sunk cost. The current reconfiguration will create a future system vulnerability. Clearly, we would like this vulnerability to be kept as low as possible.

Now we quantify this concept. Fortunately, it is quite simple to quantify future system vulnerability using

an expected value approach. We focus on a fixed processor type (e.g. UYK-44). Consider any potential configuration of processes in processors. Each processor will support one or more modes. If that processor were to fail, then the sunk cost disturbance score would be the sum of the individual mode index score plus the joint index score if any such sets are co-resident in the processor. Thus, we can compute the resulting sunk cost disturbance associated with each single processor failure. We assume that each of these processors are equal likely to fail, thus the future system vulnerability is the average of the individual total processor scores. Any configuration can be evaluated in this fashion.

3.3 Trade-offs

We have identified three aspects of the cost of a particular failure: sunk costs from a failure, the present additional cost from the reconfiguration activity and the average sunk cost of next failure (the future vulnerability). We focus entirely on a single processor type, since processes in a failed processor must be moved into the same processor type. Once a failure occurs, one can compute the sunk costs by noting all the modes that have been disturbed and adding all the individual criticality scores and joint criticality scores. We label this score S_c for current score.

We now consider a particular feasible reconfiguration. We identify the modes disturbed by failure and by reconfiguration, and calculate the total disturbance score which results. This total disturbance score is denoted as S_t. The additional disturbance from reconfiguration for this solution is $S_a = S_t - S_c$. The new configuration has a future vulnerability associated with it, the average sunk costs associated with the failure of each of the processors of this type. We label this S_f for future cost. In summary, each feasible reconfiguration has two numbers associated with it, S_a and S_f.

There are two issues which need to be addressed. Ideally, one would like to find a reconfiguration in which both S_a and S_f are small. However, these two goals are often in conflict. In addition, we have discussed a typical reconfiguration. The number of such feasible reconfigurations could be enormous, far greater than could be examined in real time. It follows that we must define heuristics which will guide us close to a solution with small values of S_a and S_f. Generally, we prefer to first specify S_a, so that the disturbance owing to reconfiguration is bounded. We then try to find a reconfiguration with a small future vulnerability S_f. The determination of an acceptable S_a requires some judgement and certainly depends on the current mission scenario. We treat the threshold limit on S_a as an input parameter to the algorithmic system. Generally, if a reconfiguration can be accomplished without further disturbance, for example, by using spares, it will be automatically carried out. If some disturbance is necessary, then input from the operator is sought. The operator, informed of the current disturbance, can either specify the set of modes which can be disturbed during reconfiguration or the maximal allowable value for S_a.

4 Algorithm Development Approach

Having discussed the principles underlying the issues of disturbance and vulnerability, we now develop our algorithms. We first describe our overall two phase approach as a basis for the design of fast algorithms. We then describe the implementation of the minimization of disturbance and of vulnerability as well as the maximization of the feasibility in the context of this two phase approach.

4.1 The Two Phase Search Approach

We have pointed out that due to the communication constraints, our reconfiguration problem is of the type of *quadratic integer assignment problems* whose optimal solutions are generally impossible to obtain unless the problem size is extremely small. Thus, we must develop fast algorithms that focus their search in the promising area of the solution space.

The constraints of our reconfiguration problem can be divided into two classes: linear constraints such as CPU sycles and memory units and the quadratic constraints, bandwidth of the buses. Since a bus is one of the most reliable elements in the system, it is less likely to fail as compared with processor elements. This indicates that a promising area in the solution space is the linear subspace. Based upon this observation, we divide our search of the solution space into two phases: the linear phase and the quadratic phase. In the linear phase, we try to satisfy all the linear constraints. Once the linear constraints are satisfied, we have a candidate soution which will be checked to see if it satisfies the bus constraints. If it does, then we have found a solution. If not, we enter the quadratic phase to further minimize the communication traffic on the bus.

4.2 Linear Phase: Satisfying Processor Constraints

In the linear phase, our reconfiguration problem is of the type known as integer assignment problems. The constraints are CPU utilization, memory utilization and communication port utilization. Note that the port utilization is additive. There are generally three possible formulations to this problem.

1. Formulate the problem as an optimal zero-one integer programming problem and use the branch and bound method.

2. Formulate the problem as the linear programming problem of the "cutting stock" type [Chvatal83].

3. Formulate the problem as a multi-dimensional bin packing problem [Coffman83].

Each of these formulations can be used. However, we prefer the bin packing formulation because of the existence of the *first fit decreasing* (FFD) algorithm, which is known to be very fast and often produces either optimal or near optimal results [Frederickson80, Bentley83]. The development of our algorithm for the search of the linear subspace will be based upon the FFD algorithm.

In the single dimension case, the FFD algorithm is to first order the items in decreasing size. We then try to

put the largest item into the first bin. Next, we try to put the next largest one into the first bin and so on until the first bin cannot be filled. We then use the second bin and repeat the same algorithm. If the bin sizes are different, we order the bin in increasing order according to their sizes. The idea of FFD is to get the most difficult one done first: trying to put the largest item into the smallest bin first. When the problem is multi-dimensional, we use the largest number of requirements to order items and the smallest number of its capacities to order bins. For example if a process (item) needs 20% CPU and 10% memory then its scalar measure is 0.2. If a processor (bin) has 70% CPU available and 30% memory available for reconfiguration task, then 30% is the scalar measure.

4.3 Quadratic Phase: Minimizing Communication Traffic

When the bus communication constraints are violated, we enter the quadratic phase. In this phase, our objective is to minimize the traffic flow on the bus. Our quadratic integer assignment problem can be formulated as follows:

1. Algorithms based upon the Branch-and-Bound method [Hillier80].

2. Algorithms based upon the estimation method [Graves70].

3. Algorithms based upon the cluster idea used in real-time task allocation [Chu80].

In a real-time environment, the only feasible algorithms are the clustering algorithms because either the Branch-and-Bound or the esimation method are known to be slow. The clustering algorithms are based upon the following observation. If we first group the heavily comunicating tasks together before assigning them to the processors, then the solution space is significantly reduced. One can argue that this reduced space is promising because many heavily communicating processes are already bound together and they are no longer to communicate over buses.

4.4 Controlling Disturbance

We have mentioned that we are in favor of keeping the reconfiguration disturbance S_a to a low level while trying to produce a reconfiguration that is low in future vulnerability. We assume that the default value of S_a is zero. That is, without the operator's explicit permission, the only processes that are allowed to be relocated by the reconfiguration system are those associated with the modes disabled by the hardware failures. The reconfiguration system will always attempt to restore disabled modes at zero disturbance and report the results to the operator. If all disabled modes cannot be restored at a zero S_a score, the operator has the option of explicitly naming the set of modes that can be relocated or of giving a level of allowable reconfiguration disturbance S_a. If S_a is given, then the reconfiguration system will first try to enlarge the solution space as much as possible, while keeping the disturbance level below S_a.

An example may help to illustrate these ideas. Suppose that we consider five modes, two associated with scanning and three with fire control. The scanning modes have individual criticality score 0.1 and joint

criticality score 0.8. The fire control modes have individual criticality score 0.15 and joint criticality score 0.9. In reconfiguration, suppose that one of the scanning modes is the only mode in the failed processor and that the operator specifies $S_a = 0.6$. If we try to relocate the functioning scanning mode, then we have a disturbance score S_a given by 0.1 + 0.8 = 0.9. The 0.1 is the individual criticality score for disturbing the functioning scanning mode. The 0.8 is the joint criticality score for disturbing both scanning modes, one of which was disturbed by the processor failure. Since 0.9 > 0.6, we must not disturb the only functioning scanning mode in the reconfiguration activity. If we relocate two fire control modes, the disturbance score is 0.15 + 0.15 = 0.3. If we relocate all three fire control modes, the disturbance score is 0.15 + 0.15 + 0.15 + 0.9 = 1.35. In this example, we can only relocate the scanning mode disabled by failure plus two functioning fire controlling modes.

Generally, once the allowable disturbance level for reconfiguration S_a is given, we need an algorithm to identify the set of modes that can be moved without exceeding the specified value of S_a. It is possible that this computation has already been carried out off-line and results are stored in the database. Thus, for a given S_a, we would only need to look up the movable modes in the database. We also provide an on-line algorithm to carry out this activity. This algorithm is a modified form of the FFD bin packing algorithm called the *reverse FFD* algorithm, which takes a set of failed modes and identifies a set of functioning modes that can be disturbed. The identified set satisfies the disturbance constraint S_a and is chosen to offer a high probability of finding a feasible reconfiguration. This algorithm is described in Section 2.3.1.

4.5 Controlling Vulnerability

We have developed our approach to keep the reconfiguration disturbance below a permitted level S_a. Given that this constraint is satisfied, we now must minimize the future vulnerability. As discussed before, we cannot hope to find the minimal future vulnerability configuration in real time. However, we can take advantage of the characteristics of the SubACS system and develop an approach that produces nearly optimal results.

When a processor fails, all the modes relying on the processes running on this processor are interrupted. Thus, to minimize future vulnerability, a key principle is to avoid putting multiple modes on a single processor, especially jointly critical modes. Another important aspect in minimizing future vulnerability is to try pack all the processes of a mode into as few processors as possible, because when a mode is supported by several processors, the failure of any of these processors interrupts the mode. Both of these aspects can be accomplished by modifying the multi-dimensional FFD algorithm in the linear phase of the reconfiguration procedure and the clustering algorithm in the quadratic phase.

In the linear phase, to avoid putting multiple critical modes into a single processor, we associated pseudo resources with each processor. Each of the pseudo resources corresponds to a joint critical index as discussed before. In the example we discussed earlier, the two scanning modes have a joint criticality score of 0.8. To prevent the processes of both scanning modes being put into a single processor, we associate one unit of

pseudo scanning resource with each of the processors that can support scanning modes. In addition, we specified that a scanning mode requires 0.6 unit of the pseudo scanning resource, while other non-scanning modes requires none of this pseudo scanning resource. Thus, a processor can only support the processes of one of the two scanning modes, but not both. To put the processes of a mode into as few processors as possible, we first run the multi-dimensional FFD algorithm for the most significant mode with respect to the processors with most available capacities. This gives the best chance for the FFD algorithm to pack the most significant mode into the least number of processors. After packing the most significant mode, we repeat this processes for the second most important mode and so on.

In the quadratic phase, the modification of the clustering algorithm is similar to the modification of the FFD algorithm. To avoid putting processes belonging to a set of jointly critical modes into a single processor, we require the clustering algorithm to observe the pseudo resource constraints associated with each processor. To promote the clustering of processes belonging to a same mode, we run the clustering algorithm on a mode by mode basis. That is, we examine the communication graph of each mode and try to cluster those processes with heavy inter-process communication first. Since processes belonging to the same mode co-operatively carry out a task, they tend to communicate with each other more than with processes of other modes. Thus the mode by mode clustering is likely to produce results nearly as good as the results of global clustering.

5 The New Algorithmic System

In this section, we first outline in detail the reverse FFD algorithm, the modified FFD and clustering algorithms. Next, we present our overall procedure for carrying out the reconfiguration.

5.1 Maximizing Feasibility: The Reverse FFD Algorithm

The Reverse FFD algorithm is used to identify the movable modes with respect to the given allowable disturbance level S_a. That is, the algorithm takes the current configuration and the set of failed modes and produces a set of functioning modes which can be disturbed within the disturbance constraint S_a.

We now list the steps of this algorithm.

1. If S_a is zero, then list all the modes that have been disabled by the failure as movable modes and exit this procedure.

2. If S_a is non-zero, then create a pseudo bin whose size is $S_t = S_c + S_a$, where S_c is the current disturbance due to the failure.

3. For each mode, we compute the average process resource utilization for each resource type. For example, mode A has 3 processes which use 0.1, 0.2, and 0.3 units of CPU and 0.4, 0.5, 0.6 units of memory respectively, then in this case the average CPU utilization is $(0.1 + 0.2 + 0.3)/3 = 0.2$ units and the average memory utilization is 0.5 units.

4. For each of the failed modes, we pair it with each of the functioning modes and check their *joint compatibility*. If the sum of the average utilization of each of the resource types is less than one,

[2]then the two modes are said to be compatible. For example, if mode A has average CPU and memory utilization 0.3 and 0.5, while these average are 0.4 and 0.3 for mode B, then mode A and mode B are said to be compatible. This is because the total utilization are 0.7 and 0.8, which are both less than 1. If a functioning mode is compatible with any of the failed modes, then it is said to be a candidate movable mode. The idea of compatibility is that if a failed mode is CPU intensive, then we want to match it with a mode which uses little CPU, so that they can be packed together later. We identify all the candidate movable modes.

5. For each candidate movable mode M_i, we compute its total resource supply R_i by summing up all the resources it utilizes in percentage of total units.

6. We now try to fill the pseudo bin by first putting all the failed modes into it. We then repeat the following steps until the list of candidate modes is emptied.

   a. For each of the candidate movable modes M_i, we compute the additional disturbance score D_i, which would arise if mode M_i is disturbed.

   b. For each of the candidate movable modes M_i, we compute the merit score R_i/D_i. The merit score is a measure of the amount of resource provided by a mode per unit disturbance incurred.

   c. We try to put the one with maximal merit score into the pseudo bin. Call this mode as the candidate bin resident and delete it from the candidate movable mode list.

   d. Add up the total disturbance scores of the bin resident modes and this candidate bin resident. If this total score is less than S_t, then the candidate becomes a resident of the pseudo bin.

7. Report all the modes in the pseudo bin as movable modes to the operator for approval and possible alteration.

5.2 The New Multi-dimensional FFD algorithm

The steps to carry out the new multi-dimensional FFD algorithm are as follows.

1. Augment each processor with a set of pseudo joint criticality resources, each of which corresponds to a joint criticality index.

2. List all the movable modes in decreasing order according to their linear criticality score.

3. List all the processors in decreasing order according to their minimal capacities. For example, if processor A has 10% available CPU cycles and 20% available memory, then the minimal capacity is 10%.

4. Take the first mode in the list and repeat the following until either the list is emptied or the reconfiguration fails.

   a. List all the processes in the mode in decreasing order according to their maximal real resource requirement such as CPU, memory, ports etc. For example if a process needs 20% CPU and 10% memory, its maximal resource requirement is 0.2.

---

[2]One is essentially good [illegible] for the first-fit bin-packing case. However, it is also for the [illegible] essentially tuned

b. Take the first process in the process list and repeat the following until either the list is emptied or reconfiguration fails.

   i. Try to put the process in the first processor in the ordered processor list.

   ii. Check all the constraints in all the dimensions, real or pseudo.

   iii. If all the constraints are satisfied, then this process becomes the resident of the processor. In this case, delete the process from the list.

   iv. If the process cannot be fit into any processor then report to operator that reconfiguration fails.

c. If the list of processes is emptied, then delete the associated mode from the list of movable modes.

5.3 The New Clustering Algorithm

The steps of the new clustering algorithm are as follows.

1. List all the movable modes in decreasing order according to their linear criticality score.

2. List all the processors in decreasing order according their minimal available capacities as defined in the previous section.

3. Take the first mode in the list and repeat the following until either the list is emptied or the reconfiguration fails.

   a. Draw the communication graph of all the processes in the mode.

   b. Identify the maximal arc.

   c. Put one of the two nodes joined by the arc on a merging list.

   d. In the graph, mark the node on the merging list as the merged node.

   e. Repeat the following until all the nodes in the graph are put into the list.

      i. In the graph, identify the node which communicates most heavily with the merged node.

      ii. Label the identified node as the candidate node.

      iii. List the candidate node in the merging list.

      iv. In the graph, merge the candidate node with the merged node.

   f. Take the first processor in the processor list. Repeat the following until the merging list is emptied.

      i. Take the first node in the merging list, delete it from the list and try to fit it in the processor.

      ii. Check all the constraints, real or pseudo.

iii. If successful, then this node is a resident of the processor.

iv. If a node becomes a resident of a processor, delete it from the communication graph.

g. Delete the processor from the processor list.

h. If the processor list is emptied but the communication graph is not, then report that the reconfiguration fails.

i. If the processor list is not emptied but the communication graph is, then delete the mode from the movable mode list.

6 The Reconfiguration Procedure

In this section, we first list the reconfiguration routine. Next, we list the overall reconfiguration procedure.

6.1 The Reconfiguration Routine

We list the reconfiguration routine.

1. Run the reverse FFD algorithm to obtain the movable mode list.

2. If it is a processor failure, then run the new multi-dimensional FFD algorithm. If this FFD algorithm fails, then report that the reconfiguration fails. In addition, report the modes that are still disabled and exit this routine.

3. Check the communication constraints.

4. If the communication constraints are satisfied, report to the operator that the reconfiguration is successful, and exit this routine.

5. Otherwise, run the clustering algorithm. If successful, report to the operator that the reconfiguration is successful, and exit this routine. If unsuccessful, report that the reconfiguration fails. In addition, report the modes that are disabled.

6.2 The Overall Reconfiguration Procedure

When a failure occurs, do the following:

1. Report to the operator the modes disabled by the failure.

2. Try to use spares if there are any. If this is successful, report to the operator that the reconfiguration is successful.

3. If this fails, run the reconfiguration routine at zero level disturbance and report the result to the operator.

4. Let the operator determine if there is any mode that can be shed and what should be the allowable disturbance threshold level, or the set of the movable modes.

5. Run the reconfiguration routine accordingly and report the result.

7 Conclusion

Dynamic system reconfiguration algorithms are important to the survivability of a tactical decision system that must operate in hostile and ever changing environments. We have sucessfully developed a system of algorithms that

1. minimize the disturbance to the functioning tasks in the course of reconfiguration,
2. reduce the vulnerability to future system failures,
3. and operate in real-time.

4.5 Algorithmic Processor Scheduler

Algorithmic Processor Scheduler

E. Douglas Jensen, John P. Lehoczky

Martin S. McKendry, Lui Sha, and Samuel Shipman

1. Processor Scheduling

In the course of investigating real time operating systems, we have become aware of a problem in many existing real-time executives that can have profound impact on the availability, maintainability, and extensibility of the system. The problem is that in these systems there is essentially no real-time system-level support for the management of time in processors in order to meet hard real-time deadlines.

Building systems that can perform demanding tasks with hard real-time deadlines is very difficult. This is generally recognized by people working in the field. For example, A. K. Mok [Mok 83] described the process of building such a system as follows:

> Current practice in designing systems for the hard real-time environment is rather *ad hoc*. There are few tools to verify that timing constraint specifications can invariably be satisfied. In fact, systems are often built with little provision for guaranteeing that stringent timing specifications can be met. Performance evaluation is accomplished either by stochastic simulation or by actual measurements on prototype testbeds, and system performance is improved by fine tuning certain system parameters. When specifications cannot be met, structural modifications may become necessary so that at least the major performance objectives can be achieved. While this iterative approach seems to suffice for building the less critical data processing systems, it is not suitable for systems which must function in a hard real-time environment. Unless the interactions among different components of a system are well understood and taken into account in the design process, there is no easy way to simultaneously satisfy multiple stringent timing constraints by fine tuning. Furthermore, there is no absolute guarantee that a timing constraint will invariably be met by the use of stochastic simulation or by taking performance measurements for a limited set of load conditions. For this reason, most current systems are better categorized as *soft real-time* in that they lack hard guarantees on vital performance characteristics.

One reason this approach has been taken is that the theoretical results necessary to do better are not widely known, and in some cases are new. In fact, many aspects of the problem are active areas of research. Furthermore, designers of real-time operating systems have traditionally been content to provide a set of primitive operations with which to perform scheduling, and leave the problem of actually building a scheduling structure to the applications programmers, with predictably *ad hoc* results.

Given the tools they have had to work with, programmers have done surprisingly well. But those tools are inadequate for dealing with demanding hard real-time environments. In complex distributed systems, the need is urgent for a unified system-level scheduling strategy.

In this report, we first discuss the problems with the existing approach and point out the important benefits to be gained from an algorithmic scheduler. We then give a tutorial on the basics of scheduling theory and issues to be addressed in developing an algorithmic scheduler. Further work on the problem is presented in subsequent reports in this volume.

1.1 Problems with Existing Approaches

In this section, we describe a hypothetical scheduling facility that is similar to most existing ones in real time executives. Generally, a scheduling facility supports the notion of *programs* which are composed of *tasks*. Tasks are the schedulable entities. Tasks within a single program can communicate, synchronize, and alter the priority of other tasks using system services. However, no task can directly affect a task in a different program except by using the interprocess communication facilities. The set of tasks running on a particular processor is termed a *processor load*.

Tasks have two possible dispatching states: *ready to run* and *blocked*. Each task also has a numerical priority When the dispatcher runs, it dispatches the highest-priority task that is ready to run. A new task is run when the currently running task becomes blocked (typically by waiting for some system service) or when some higher-priority task becomes ready to run (e.g., upon I/O completion). There are system calls to create and destroy tasks, and to alter the priority of any task.

Priorities are assigned to the tasks within the loads on the basis of system simulation results or in whatever way the individual software developers see fit. No global policy is used to determine these priorities. Apparently, they are based on the programmer's perception of the importance of the task relative to other tasks in the same program. Some of the ideas such as giving higher priorities to I/O activities are borrowed from techniques to maximize throughput in time sharing systems. The essential problem with ad hoc priority assignment methods is the lack of understanding of the relationship between throughput, task importance, and task deadlines.

After assigning priorities to tasks, each load is then individually tested. If one fails to meet any of its deadlines, an attempt is made to modify it so no deadlines are missed. A modification may involve adding or deleting one or more programs from the load, but usually less drastic measures are attempted first. Such measures may involve changing task priorities, changing program logic (e.g., the order of certain events, or synchronization), making performance improvements, etc. The programs in the load are thus modified until they pass the tests.

This process, which we will refer to as "hand-crafting" processor loads, must be performed for each processor load, and may have to be repeated when changes are made to the software—even for changes that seem unrelated to scheduling.

If the tasks in a particular processor load are not co-schedulable (i.e., there are conditions under which some task will be forced to miss its deadline), it is hoped that these conditions will become apparent during system simulation or during testing, so they can be dealt with as described above. Failing that, it is hoped that if these conditions do not occur during testing, they will also not occur while the system is in actual use. Unfortunately, such conditions are most likely to occur in situations in which the system is under stress, such as battle conditions.

1.1 Problems with Existing Approaches

In this section, we describe a hypothetical scheduling facility that is similar to most existing ones in real time executives. Generally, a scheduling facility supports the notion of *programs* which are composed of *tasks.* Tasks are the schedulable entities. Tasks within a single program can communicate, synchronize, and alter the priority of other tasks using system services. However, no task can directly affect a task in a different program except by using the interprocess communication facilities. The set of tasks running on a particular processor is termed a *processor load.*

Tasks have two possible dispatching states: *ready to run* and *blocked.* Each task also has a numerical priority When the dispatcher runs, it dispatches the highest-priority task that is ready to run. A new task is run when the currently running task becomes blocked (typically by waiting for some system service) or when some higher-priority task becomes ready to run (e.g., upon I/O completion). There are system calls to create and destroy tasks, and to alter the priority of any task.

Priorities are assigned to the tasks within the loads on the basis of system simulation results or in whatever way the individual software developers see fit. No global policy is used to determine these priorities. Apparently, they are based on the programmer's perception of the importance of the task relative to other tasks in the same program. Some of the ideas such as giving higher priorities to I/O activities are borrowed from techniques to maximize throughput in time sharing systems. The essential problem with ad hoc priority assignment methods is the lack of understanding of the relationship between throughput, task importance, and task deadlines.

After assigning priorities to tasks, each load is then individually tested. If one fails to meet any of its deadlines, an attempt is made to modify it so no deadlines are missed. A modification may involve adding or deleting one or more programs from the load, but usually less drastic measures are attempted first. Such measures may involve changing task priorities, changing program logic (e.g., the order of certain events, or synchronization), making performance improvements, etc. The programs in the load are thus modified until they pass the tests.

This process, which we will refer to as "hand-crafting" processor loads, must be performed for each processor load, and may have to be repeated when changes are made to the software—even for changes that seem unrelated to scheduling.

If the tasks in a particular processor load are not co-schedulable (i.e., there are conditions under which some task will be forced to miss its deadline), it is hoped that these conditions will become apparent during system simulation or during testing, so they can be dealt with as described above. Failing that, it is hoped that if these conditions do not occur during testing, they will also not occur while the system is in actual use. Unfortunately, such conditions are most likely to occur in situations in which the system is under stress, such as battle conditions.

This method of system integration has profound impacts on many aspects of the system. We explain these impacts below.

1.1.1 System Integration

At system integration time, each processor load is hand-crafted so that it will meet its real-time constraints. This can be a very time-consuming process. Timing-dependent problems, especially transient ones, are the hardest bugs to find in a computer system. In the absence of a scientific methodology for managing task priorities, the current practice can only be characterized as an *ad hoc*, trial and error approach, depending on the skill, intuition, and luck of the system integrator for any chance of success.

Transient timing problems are bound to occur in such a system, due to the low-level nature of the dispatching primitives used to effect scheduling and the lack of an operating-system level scheduler. Any change in the timing environment can perturb such a system, including but not limited to changes in network activity or operating system executive activity. (These dependencies are at the root of a wide spectrum of timing problems, and could be largely eliminated by implementing a processor scheduler. See Section 1.2.) Fixing these timing problems, as well as problems with the functionality of the software (as opposed to its timing) will require changes to the programs.

Of course, changes to any program could potentially affect the schedulability of every processor load of which that program is a part. It is even conceivable that a change in one program could affect processor loads of which it is *not* a part, due to its possible effects on network activity, which could ripple through the system, given the sensitivity of the current scheduling methods to such factors.

So, in the event of any change (and changes are relatively frequent at this stage of the system life-cycle) the time-consuming process of testing and debugging the *timing* of every processor load of which that program is a part is begun again. And every load that fails to pass the tests must go through the hand-crafting cycle again to make sure it satisfies the scheduling requirements.

There are certain trade-offs which can affect the seriousness of the problem at system integration time. One involves utilization. The higher the degree of utilization at which processors are to be run, the more likely that a given processor load will not be schedulable. This means that more exercises in hand-crafting will be required, and that these exercises will be more difficult and more time-consuming to successfully complete. The alternative is to run at lower utilization levels, sacrificing a large degree of system availability and fault-tolerance.

Another trade-off concerns flexibility in system reconfiguration, which also directly affects system availability. The more varied processor loads that are available, the more flexible the reconfiguration process will be, and the better the system will be able to recover from hardware failures. But if developing and testing

each processor load is an expensive process, then there will be pressure to produce fewer processor loads. This will reduce reconfiguration flexibility.

The current handcraft approach also significantly impacts the testability of system timing. A major aspect of testability is the observability of system state. In 68K-OS, the information needed to schedule tasks is spread out over all the tasks in a processor load, making it difficult to access during debugging. In a system with a scheduler, the scheduling information would be localized within the scheduler, in its proper context, and easily accessible. Another important aspect is simply the complexity of the system. The operation of scheduling, as we have seen, is considerably more complex than it would be under an operating system with a scheduler. This complexity makes the system more difficult to test and debug.

Using the handcraft approach, the amount of testing that must be done to establish confidence in the system is quite large. In addition to testing the *functionality* of the system, the *schedulability* of the system must be continually tested as well. If there were a processor scheduler common to all the software in the system, confidence in the ability of that scheduler to correctly schedule the system would be built up incrementally with each test. Furthermore, each test of each program would present an opportunity to discover bugs in the *one* piece of software concerned with scheduling, namely the scheduler.

As it is now, a new scheduler, in effect, is devised for every processor load. Extra testing must be done to test both this "scheduler" and the programs it schedules. This schedulability testing can be more time-consuming than testing the functionality of the code, since timing problems are generally more difficult to detect, find, and fix than other types of bugs.

Of course, this is not to imply that a better scheduling facility would eliminate the need for acceptance testing of the system. But since no amount of testing will absolutely guarantee the absence of bugs, and only a finite amount of testing can be done, then it is better to allocate the available testing time to best advantage. Having a scheduler frees much of the time the old method would have used for testing the schedulability of every processor load, and allows it to be used to test other aspects of the system.

1.1.2 Post-Deployment Support

The concerns about the impact of system scheduling in the system integration phase become even more important after the system is deployed. Actual use of the system will inevitably reveal changes that need to be made, either to fix problems with the software, or to enhance its functionality or performance.

Consider what happens when a single program in a processor load is changed. Any change is likely to alter the timing behavior of the program. The new behavior will then be different from the behavior it displayed during system testing. Thus each processor load of which it is a part must be tested again, not to test the functionality of the change—one processor load test would suffice for that—but to test the schedulability of

each of the processor loads. It is possible that the change would render some subset of the loads unschedulable, in which case the cycle of "hand-crafting" would be entered again, as described above.

Of course, the same applies to any hardware upgrade that alters the timing behavior of the hardware. Performance upgrades fall into this category. Thus, if more capable processors were to be used, the entire set of processor loads would have to be reformulated in order to take advantage of the added performance. If two versions of a processor, different only in speed, were to co-exist in the system, two sets of processor loads would have to be developed if the performance advantage of the faster processors were to be exploited.

As we mentioned above, this relatively expensive cycle of changes occurs fairly often during system integration, and must be tolerated at that point in order to finally arrive at a working system. Once the system is working (for some suitable definition of "working") and is deployed, the cost of any change rises dramatically.

The point is that upgrades, of either software or hardware, can become very difficult when the current scheduling methods are used. In fact, since timing is the most difficult part of a real-time system to get right, there would be pressure to avoid any changes that could introduce timing problems, given that the means of detecting and dealing with them are so unwieldy. There are already instances of this. A. K. Mok [Mok 83] cites the following news item about the F18 aircraft appeared in *ACM SIGSOFT Software Engineering Notes* [SEN 81]: "Apparently the effort that has gone into developing and testing the software is so extensive that, when changes are required, it is now preferable to modify the plane to fit the existing software, rather than to modify the software to match the plane." Glenford Myers [Myers 76] states that the costs of maintenance and testing account for roughly 75% of the cost of software, and cites several examples, including the SAGE system which cost $20 million *per year* after 10 years of operation, compared to its initial development cost of $250 million. A study by B. W. Boehm of TRW Corporation indicated that the life-cycle cost of real-time software products has been three times as much as that for analytical software products.

1.2 The Solution: A Processor Scheduler

1.2.1 Proposed Solution

We propose that implementing a processor scheduler as part of the operating system is a significant step toward addressing the problems we have discussed. The processor scheduler should implement a proven scheduling policy. It should require that the applications provide accurate information on which to base scheduling decisions, but will otherwise be independent of the application code. The rationale for such a scheduler and its potential benefits are discussed below.

1.2.2 Rationale and Benefits

It is evident that *time* is an important resource in a real-time system. The management of time is as important as the management of databases or communications. Yet there is no on-line system-level support for the management of processor time. Managing time involves, among other things, scheduling traffic on the communications bus, scheduling processes to run on processors, and coordinating across processors. A system-level strategy for managing time is needed, and a processor scheduler is a step toward that strategy.

We have seen that a processor scheduler is needed to implement scheduling policies. Theoretical work on scheduling shows that certain policies are optimal in given circumstances, and indicates the possibility, with further work, of developing processor scheduling algorithms that are efficient and, perhaps more importantly, are guaranteed to correctly schedule a schedulable set of processes.

A scheduler based on a scientific scheduling policy, having the appropriate mechanisms to implement that policy, could potentially eliminate transient timing problems. This could greatly lower system testing and integration costs, since such timing errors account for a large fraction of those costs. Independence from such transient effects also enhances modularity, since such dependencies would no longer affect the applications code.

A processor scheduler also greatly enhances the modularity of the system, in the sense that separating the concern of scheduling from the concern of correctly functioning applications code implies that changes to that code be independent of scheduling issues. Either a set of processes is schedulable or not. If the scheduler determines that they are not, then a different set of processes must be used. If they are schedulable, then there is no question of "hand-crafting" them to meet deadlines.

This greatly improved modularity has profound effects on the feasibility of post-deployment support. Programs are no longer interdependent with regard to scheduling. The scheduler handles those problems, and programs can be modified without particular concern for schedulability, beyond whether enough processor resources will be available to support the enhanced program. This means that system maintenance and upgrades to software will be more straightforward. Further, a parametrized implementation of the processor scheduler would enable performance upgrades to the hardware to be incorporated without necessitating modifications to the applications software.

System modularity also improves testability of system timing, since information related to scheduling is localized in the scheduler and so is more easily accessible than if it is spread among all the applications tasks. The entire scheduling system is less complex, since it is localized in a scheduler module in the operating system, which has well-defined interfaces and interactions with applications tasks and other parts of the system. Structuring a scheduler as a single module means that it is possible to test that module as a unit. Furthermore, every test of applications software incidentally tests the scheduler as well, in contrast to the current method, in which the scheduling of each processor load is different and must be tested separately.

Fixes to problems found within the scheduler benefit all processor loads, whereas scheduling problems found in the current method only apply to the particular processor load in which they are encountered.

Aside from these considerable benefits, a processor scheduler paves the way for process level reconfiguration, a method of responding to hardware failures that has enormous advantages over the current method of processor level reconfiguration. The latter method is dictated, in part, by the lack of a processor scheduler.

Another advantage, in terms of reusable software, is that a scheduler developed for a system would be useful in other real time systems.

These goals can only be achieved if a longstanding tradition of real-time systems is broken. Heretofore, the major responsibility of the developers of real-time operating systems was seen to be providing a set of primitive operations for scheduling. These primitives were to be used by the application programmers to build their own scheduling structures as they saw fit. The operating system assumed no responsibility for schedulability, which was just as well, since most (if not all) such systems could provide no guarantees.

However, in a highly complex system this is insufficient. There is a need for a scheduling *service* to be provided to applications, which obey a discipline of using the scheduler's operations, in exchange for some guarantee of schedulability. The operating system must assume responsiblity for schedulability, because it is the logical place in the system for the implementation of a scientific scheduling policy. The operating system should provide an environment for applications software in which the scheduling issues are handled for it, just as current operating systems provide an environment for processes in which control of I/O devices is handled by the operating system on their behalf.

The current situation with respect to scheduling is analogous to the situation that prevailed in programming languages before the advent of structured programming. The programmer was provided with a powerful primitive for control structuring, the GOTO statement, and was expected to use it to build whatever control structures he saw fit. Since structured programming has been accepted by most of the computing community, and languages have emerged which support this style, the days of the GOTO-filled program are not missed by the majority of programmers. They are willing to sacrifice some degree of power for the benefits of understandability and maintainability that a structured approach to the flow of control in programs confers.

In the case of scheduling, real-time programmers should be ready to give up the power of direct control over the operating systems scheduling actions in favor of the benefits of modularity and guaranteed scheduling that an operating system-level processor scheduler confers. And, as programming language designers were charged with the task of providing useful and usable control structuring primitives, the real-time operating system designers are now charged with providing useful and usable scheduling facilities.

2. Scheduling Theory

2.1 Basic Concepts

2.1.1 Policy and Mechanism

It is useful to view a process scheduling facility as consisting of two conceptual parts: policy and mechanism. The policy determines how scheduling decisions are made, and the mechanism is the agency by which they are carried out. Policy and mechanism are directly analogous to the concepts of strategy and tactics. The policy/mechanism dichotomy has useful application to other forms of resource management as well. The Archons project of Carnegie-Mellon University contains many useful examples.

To illustrate this view, we will briefly describe the general scheduling model developed by Ruschitzka and Fabry [Rusch 77] In this model, a scheduling policy consists of three parts:

- a decision mode
- a priority function
- an arbitration rule

The decision mode determines when a rescheduling occurs. Common examples of the decision mode are non-preemptive, pre-emptive, or quantum-oriented pre-emptive (in which a task is allowed to execute continuously for at most a fixed period of time, the quantum, after which a rescheduling may occur).

The priority function is an arbitrary function of task and system parameters that assigns a priority to a task. The priority could be a function of task run time, task period, system load, etc. A priority function can be defined to support any scheduling policy including commonly used policies such as FIFO, round-robin, deadline, or rate-monotonic scheduling.

The arbitration rule resolves any conflict which may occur among jobs with equal highest priority. Examples of possible arbitration rules include FIFO and round-robin.

In this model, mechanisms would be required to support the scheduling policy. These include mechanisms to cause the scheduler to run at a time determined by the decision mode, to evaluate the priority function, and to dispatch the task indicated by the priority function or the arbitration rule.

In following sections, we will see that it is important that a scheduling facility provide good mechanisms for specifying and implementing scheduling policies. It is in this area that the current SubACS mechanisms fall short.

2.1.2 Policies and Performance

To understand the issues involved in scheduling real-time systems with hard deadlines, it is necessary to review some results from scheduling theory. These results are relatively straightforward analytically, but are not intuitively obvious and consequently may be surprising. The discussion below is based on the work of Liu and Layland [Liu 73].

2.1.2.1 Assumptions

A number of simplifying assumptions are made in this work:

- We are scheduling *periodic* tasks with *hard deadlines.*
- A task has:
  - a relative initiation time, I,
  - a period, T, and
  - an amount of processing, C, that must be completed within the period T.
- The tasks are *independent*, i.e. there are no precedence relations or synchronization among tasks.
- Tasks can be instantly pre-empted.
- No overhead is incurred in task swapping, interrupt handling, scheduling, etc.
- Priority granularity (number of distinct priority levels) is adequate.

These assumptions are admittedly unrealistic; nevertheless, they lead to interesting and useful results. Much of our work is aimed at relaxing these assumptions without losing the ability to make guarantees about the schedulability of tasks.

Under these assumptions we will consider how tasks can be scheduled to ensure that all processing is done and all deadlines are met. Within this framework, we will consider three issues:

- Can the initiation times (I) be controlled, or are they arbitrary?
- Can the periods (T) be controlled, or are they arbitrary?
- Is a fixed or dynamic priority scheduling algorithm to be used?

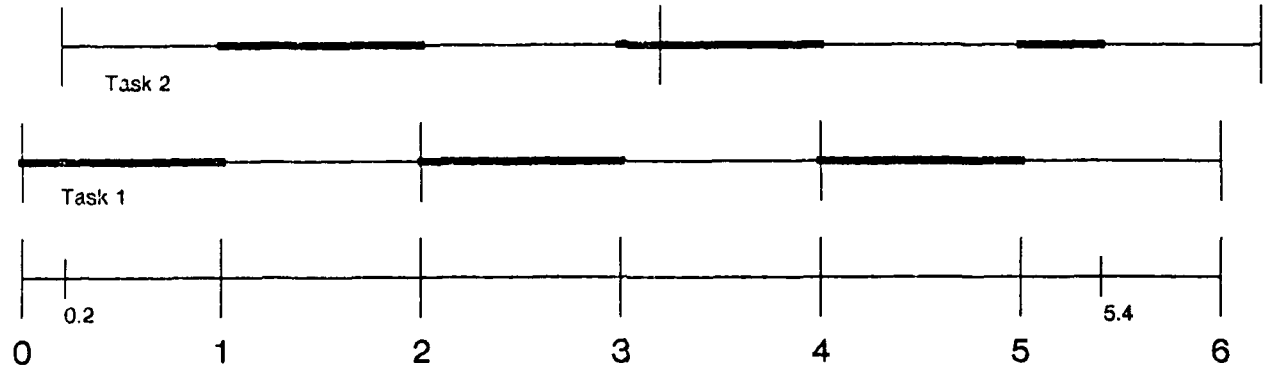
In a later section we will consider the effects of conditions which violate the assumptions, e.g. non-zero scheduling overhead and interrupt processing.

2.1.2.2 Examples

From the assumptions, if a task has a relative initiation time I and period T, then it is initiated at times $I, I+T, I+2T, I+3T, \cdots$. The processing requirement, C, for the task initiated at time $I+kT$ must be completed by time $I+(k+1)T$.

Another quantity of interest is the utilization U of the task. The utilization is a measure of how much of the processor is being used by the task. A task has utilization $U=C/T$. A set of n tasks has total utilization $U = C_1/T_2 + C_2/T_2 + \cdots + C_n/T_n$.

Consider the example in Figure 2-1. Task 1 has higher priority than task 2. Note that if task 2 is given higher priority, then task 1 will miss its deadline at time 2. This situation is illustrated in Figure 2-2.



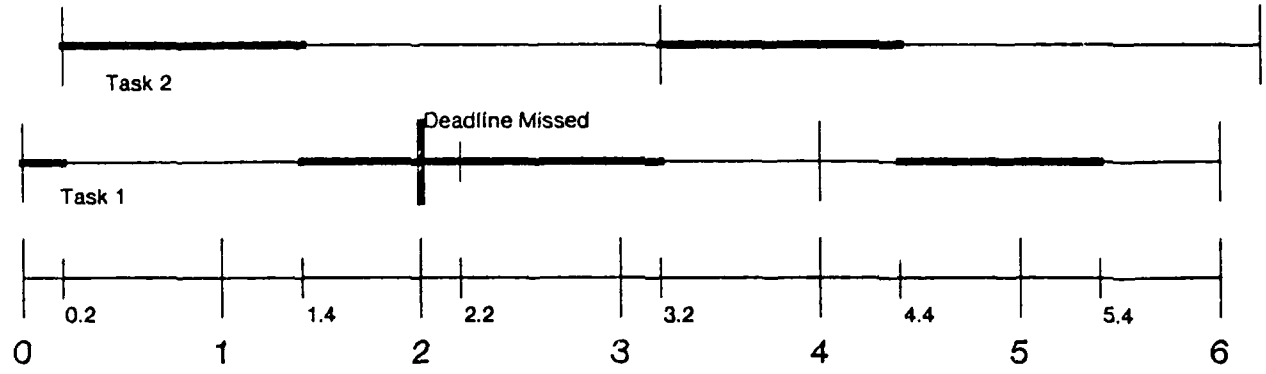
$I_1 = 0$ $C_1 = 1.0$ $T_1 = 2$
$I_2 = 0.2$ $C_2 = 1.2$ $T_2 = 3$
$U_1 = 0.5$ $U_2 = 0.4$ $U_{total} = 0.9$

Figure 2-1: Two Periodic Tasks Scheduled Correctly

Now consider the modified example in Figure 2-3. Note that task 1 has higher priority than task 2, and task 2 misses its deadline at time 3. However, if task 2 is given higher priority, then task one will miss its deadline at time 2. This situation is illustrated in Figure 2-4. So we can see that this case *cannot* be scheduled using a *fixed priority* algorithm, i.e. an algorithm that assigns an unchanging priority to each of the tasks.

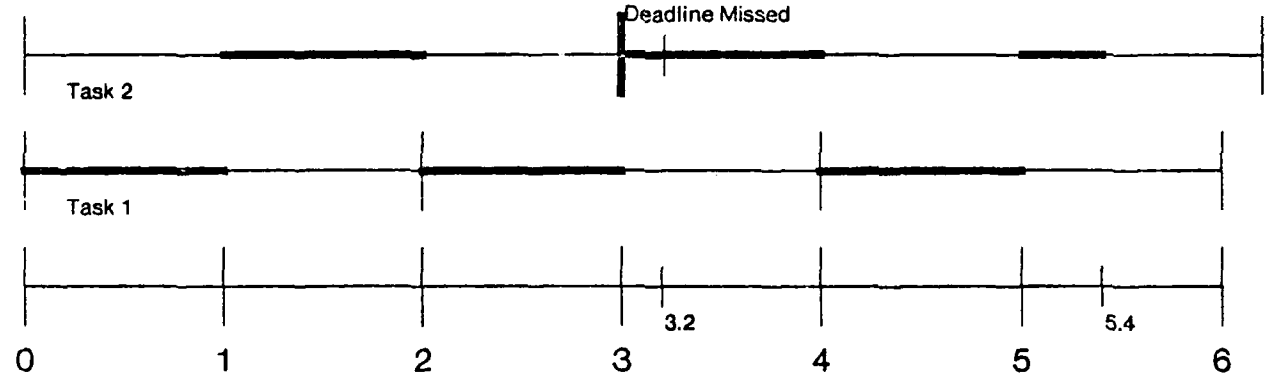
A *dynamic priority* algorithm, however, will work. In Figure 2-5 the two tasks are scheduled such that their deadlines are met by varying their relative priority. At times, one task has higher priority, while at other times the other task does. In this case, the algorithm used for assigning the priorities is known as the *deadline scheduling* algorithm, in which the task with the nearest deadline has highest priority.

Note that in this case, a fixed priority algorithm will work if $I_1 = 0$ and $0.1 \le I_2 \le 0.4$ or $0.6 \le I_2 \le 0.9$. In



$I_1 = 0$ $C_1 = 1.0$ $T_1 = 2$
$I_2 = 0.2$ $C_2 = 1.2$ $T_2 = 3$
$U_1 = 0.5$ $U_2 = 0.4$ $U_{total} = 0.9$

Figure 2-2: Tasks of Figure 2-1 Scheduled Incorrectly



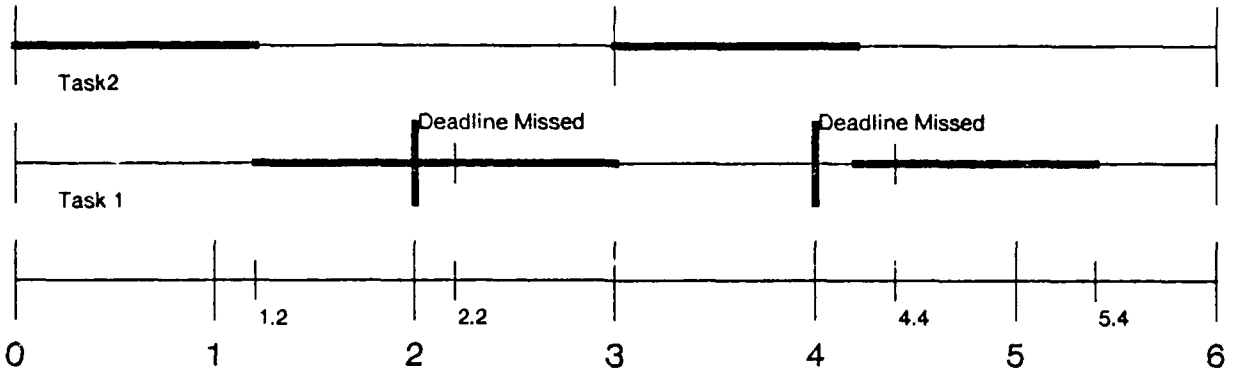
$I_1 = 0$ $C_1 = 1.0$ $T_1 = 2$
$I_2 = 0$ $C_2 = 1.2$ $T_2 = 3$
$U_1 = 0.5$ $U_2 = 0.4$ $U_{total} = 0.9$

Figure 2-3: Fixed Priority Unschedulable 1

other words, 60% of the possible phasings of the two tasks are schedulable, 40% are not. However, deadline scheduling will *always* work if the total utilization $U \leq 1.0$.

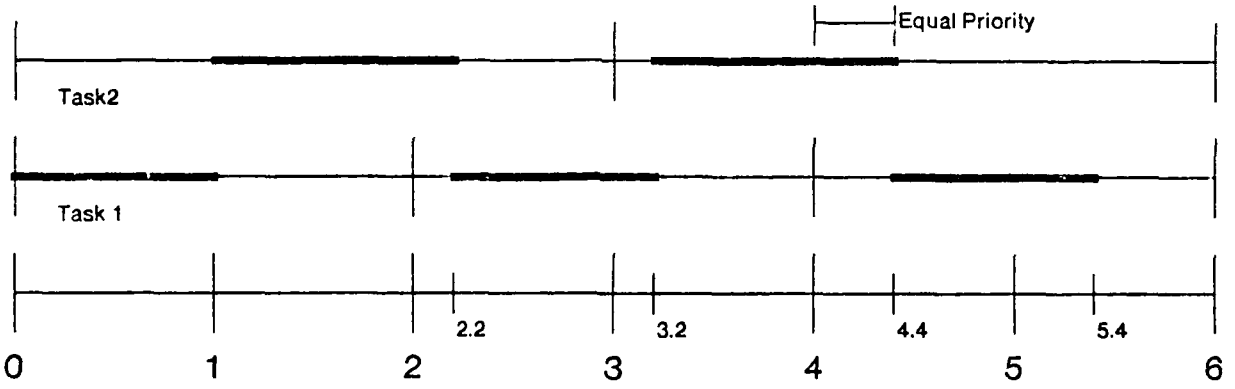
2.1.2.3 Results

The worst case for scheduling periodic tasks occurs when all initiation times are the same (equal to zero). Liu and Layland call this the *critical instant.* Since this is the worst possible case, only the first deadline of each task must be checked to determine whether the tasks are schedulable. If the first deadline after the critical instant is met, then all other deadlines will be met as well. Liu and Layland analyzed scheduling algorithms for this worst-case situation to determine the optimal algorithms for fixed priority scheduling and dynamic priority scheduling.



$I_1 = 0$ $C_1 = 1.0$ $T_1 = 2$
$I_2 = 0$ $C_2 = 1.2$ $T_2 = 3$
$U_1 = 0.5$ $U_2 = 0.4$ $U_{total} = 0.9$

Figure 2-4: Fixed Priority Unschedulable 2



$I_1 = 0$ $C_1 = 1.0$ $T_1 = 2$
$I_2 = 0$ $C_2 = 1.2$ $T_2 = 3$
$U_1 = 0.5$ $U_2 = 0.4$ $U_{total} = 0.9$

Figure 2-5: Deadline Scheduling of Tasks

For fixed priority scheduling, an optimal algorithm is the *rate monotonic* algorithm. The rule for assigning priorities to tasks in this algorithm is that task i has priority over task j if and only if $T_i < T_j$. If $T_i = T_j$, the relative priorities are arbitrary. (Conceptually, tasks i and j can be combined into a single task, which has the same effect on schedulability as the two separate tasks of equal period). Priorities are assigned based *only* on the periods. The C_i are ignored.

The rate monotonic algorithm can schedule any phasing of any task set having utilization $U \leq log_e 2 \approx .693$

(or $n(2^{1/n}-1)$), for a set of n tasks). This is the worst case.[1]

The *deadline scheduling* algorithm is a dynamic priority algorithm that is optimal in general. It can schedule *any* task set having total utilization $U \leq 1.0$.

A dynamic priority algorithm can be more difficult to implement than one that uses fixed priorities, but the results of Liu and Layland's analysis indicate that even the optimal fixed priority scheduling algorithm sacrifices over 30% of CPU utilization if schedulability is to be guaranteed. In contrast, the optimal (dynamic priority) scheduling algorithm sacrifices no CPU utilization. Therefore, it would be advantageous to use the deadline scheduling algorithm, since it is unlikely that the additional overhead it would incur could even approach the 30% already wasted by the rate monotonic algorithm.

2.2 Priority Inversion: A Serious Problem

2.2.1 Examples

Having introduced the basic concepts and results of scheduling theory, we now examine the effect of deviating from an optimal scheduling algorithm. Priority inversion occurs when a task runs at a priority different from that dictated by an optimal scheduling algorithm. The schedulability results presented earlier are no longer applicable. It is possible to find deadlines being missed with the task set having relatively small total utilization if priority inversions are allowed to occur.

Figure 2-2 is an example of priority inversion. The tasks are given priorities that are exactly the inverse of those that would be assigned by the rate monotonic algorithm, and so they are no longer schedulable. This is a particularly blatant example of priority inversion, but in fact even a "small" violation of the priority assignments can cause severe problems.

Consider the following example:

Task 1	$I_1=0$	$C_1=0.01$	$T_1=1$	$U_1=0.01$
Task 2	$I_2=0$	$C_2=1.00$	$T_2=100$	$U_2=0.01$
				$U_{total}=0.02$ (!)

In this case, if we schedule the tasks according to the rate monotonic algorithm, then task 1 has higher priority than task 2, and the tasks are schedulable. If, however, the priorities become inverted (task 2 has higher priority) then *this task set cannot be scheduled.* Task 1 will miss its deadline at T=1.

---

[1] There is an interesting best case for the rate monotonic algorithm, in which if the set of tasks is *totally harmonic*, i.e. the ratio of any period to any longer (or equal) period is an integer. The rate montonic algorithm can schedule any task set having this property, if the total utilization $U \leq 1.0$.

This example is admittedly contrived, but it serves to illustrate graphically the problems that priority inversion can cause with scheduling even the most (seemingly) innocuous task set. Priority inversion is a most serious problem since, if it occurs, there is *no lower bound* below which correct scheduling can be guaranteed.

3. Real Time Scheduling Issues

In the previous section, we have discussed the scheduling theory under idealized circumstances, introduced the concept of priority inversion and illustrated its potential harmful effects. In this section, we discuss a number of practical issues which can cause priority inversion if they are not handled properly. In addition, we give an overview of the approach used by the Archons project at Carnegie-Mellon University for solving real time scheduling problems.

3.1 Sources of Priority Inversion

In the scheduling of real time tasks, there are three important sources of possible priority inversion:

- I/O scheduling: the arbitration of DMA and interrupts.
- Task interactions: the handling of task precedence relations, of inter-process communication and of the synchronization of accesses to shared data.
- Transient overloads to processors.

We discuss each of the topics in turn.

3.1.1 I/O Scheduling

The first important source of possible priority inversion in many real time executives is the inconsistency between the algorithms used to schedule the I/O activities and to schedule tasks in the processor. To illustrate this point, let us first examine the structure of periodic tasks.

A typical periodic task consists of three phases. The first phase is the transfer of the data from an input device into the main memory. The second phase is the execution of the computation. The third phase is the transfer of the results of the computation to an output device. All three phases must be completed during the task period in order for the task to meet its deadline. Since a single cycle of a real time task includes both I/O and computation, it is easy to see that the scheduling of the I/O activities must be consistent with the scheduling of the computation activities of the processor. Nevertheless, the general practice found in many real time systems is to arbitrate DMAs using either a FIFO or a round-robin scheduling policy. At the end of a DMA, the I/O device interrupts the processor with an aribitarily defined priority level that is unrelated to the algorithm used to schedule the processor. Such a practice is a source of priority inversion, because the I/O activity assocated with a low priority task can slow down either the computation or the I/O activity of a high priority task. To exemplify the potential problem caused by this practice, let us condsider the following example.

Suppose we have 5 tasks, T_1, ..., T_5, with periods 10, 20, 30, 40, and 50. Each task has 2 time units for DMA-in, 2 time units for computation and 2 time units for DMA-out. For simplicity, we further assume the

"cycle stealing" or processor slowing down effect of DMA and the overhead of handling DMA completion interrupts are negligible. Suppose that at time zero, all the 5 tasks are ready to have their data input by DMAs. Suppose the data input to task T_1 has arrived somewhat later than others. If we follow the FIFO rule for executing DMAs, then task T_1 cannot start until t = 10 and consequently misses its deadline. On the other hand, if we use the rate monotonic scheduling algorithm to schedule the computations *and* to schedule DMAs, then the data are input in the order of task T1 to task T5 and all the deadlines are met. Thus, in a real time operating system, it is important to arbitrate I/O activities in an order that is consistent with the processor scheduling algorithm.

3.1.2 Task Interactions

In many real time operating systems, the second important source of potential priority inversions is the method of handling task interactions.

Tasks can have precedence relations and can share data. In addition, tasks can interact with each other through system executive calls or inter-process communication (IPC) mechanisms. In most existing real time systems, the mechanisms to handle task interactions are those developed in the context of time sharing systems, such as semaphores for synchronization and mailboxes (or ports) for IPC. These mechanisms were developed without any consideration of satisfying real time constraints. For example, an undisciplined use of semaphores could allow a low priority task to block the execution of a high priority task. Indeed, in many existing systems the operating system executive executes at the highest priority and the execution on behalf of a system call executes at this highest priority. Consequently, a prolonged system call can lock out other tasks that might have become the highest priority task during that time period. In a real time operating system, mechanisms for synchronization, IPC, and system executive calls must be structured in a way that complements rather than interferes with the scheduling activities.

3.1.3 Transient Processor Overloads

The third source of potential priority inversion results from attempts to protect important tasks from missing deadlines under periods of transient processor overload. Transient processor overload can be caused by stochastic processing time or exception handling. In some applications, the computation time of a task is highly data dependent. In some other cases, the exception handling codes are large but invoked so infrequently that it is not wise to reserve CPU cycles for them during periods of normal execution. In either of the two situations, in order to provide an adequate average processor utilization, one must be prepared to tolerate some occasional transient processor overload.

There is the tendency for programmers to assign a higher priority to a task that is more important to a mission than other tasks. This tendency is especially pronounced when there is the possiblility of transient processor overloads. However, in real time systems it is crucial to meet *all* the deadlines whenever possible. Assigning higher priority to a more important task without analyzing the impact on deadlines could force less

important tasks to miss their deadlines unnecessarily. For example, suppose that task 1 has a period of 5 time units and an execution time of 2 units, while task 2 has a period of 10 units and execution time 4 units. Task 2 is also known to be more important than task 1. According the the rate monotonic algorithm, task 1 should be given higher priority, and both tasks can be successfully scheduled using this algorithm. If task 2 were assigned higher priority, than task 1 would be forced to miss its deadline unnecessarily.

On the other hand, the importance of the task becomes crucial when the processor is sufficiently overloaded, so that all the deadlines cannot be met. In an overloaded situation, we want to sacrifice the less important tasks. It is important to observe that the simple rate monotonic or deadline scheduling algorithms do not behave well under overloaded conditions. In the case of the rate monotonic algorithm, the tasks with longest period will be the ones to miss their deadlines. In the case of dynamic deadline scheduling algorithms, any or all of the task could miss its deadline during an overload period.

In summary, the problem of transient processor overloads is a serious one. The practice of assigning higher priority to a more important task tends to force less important tasks to miss their deadlines unnecessarily. On the other hand, simple scheduling algorithms such as rate monotonic or deadline scheduling algorithms cannot guarantee that the deadlines of important tasks will be met during an overload situation. Nonetheless, schedulers developed for real time operating systems should ensure that the deadlines of all tasks will be met when the system is not overloaded. Furthermore, only those least important tasks' deadlines should be missed when the system is experiencing an overload.

References

[Liu 73] Liu, C. L. and Layland, J. W.
Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment.
JACM, 1973.

[Mok 83] Aloysius Ka-Lau Mok.
Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment.
PhD thesis, Massachusetts Institute of Technology, 1983.

[Myers 76] Myers, G.
Software Reliability.
John Wiley & Sons, 1976.

[Rusch 77] Ruschitzka and Fabry.
A Unifying Approach to Scheduling.
CACM 20(7), 1977.

[SEN 81] Staff.
News Item.
ACM SIGSOFT Software Engineering Notes 6(2), 1981.

4.6 Analysis of Processor Scheduling with Overhead

Analysis of Processor Scheduling with Overhead

1 Introduction

An analysis of the schedulability of periodic tasks with hard deadlines was presented by Liu and Layland [Liu 73]. This analysis indicates that any such task set can be scheduled if it utilizes no more than the available processor cycles. Unfortunately, this analysis ignored several factors that are important in many systems. These include I/O and timer interrupts which can cause priority inversions, the overhead of the scheduling activity itself and overhead from task swapping. Some of these can be regarded as a part of the periodic tasks and so can be incorporated into the analysis. Other overhead aspects cannot be so regarded. In general, we must develop an analysis which includes a variety of overheads.

2 A Quantum Formulation

There are several ways to deal with the interrupt problem. First, one could have a co-processor which would handle the scheduling and interrupts. This would leave only task swapping, and it could be incorporated directly into the scheduling analysis. A second approach is to create a time quantum during which a task can be executed without interruption. In this approach, some hardware queueing device would be needed to queue the interrupts arriving in the middle of a quantum. We will analyze this latter approach in this section and develop queueing/co-processor solutions in a subsequent report. We must, however, determine the power of dynamic deadline scheduling or rate monotonic when overhead is present and a quantum approach is used. There is an important trade-off issue concerning the quantum length. On the one hand, a long quantum allows a high priority task to be executed without interruption and the attendant priority inversions. On the other hand, a long quantum allows a low priority task to execute for a long period without interruption as well. This is likely to cause a priority inversion as higher priority tasks are forced to wait. In such a fixed quantum size situation, one can determine the optimal size to allow maximum utilizations. We will see, however, a variable quantum length is feasible and has a greater potential.

3 A Fixed Quantum Formulation

The processor scheduling problem has the following formulation. There are n periodic tasks. The i^{th} task has period T_i and requires C_i units of processing time. Let $U_i = C_i/T_i$ be the utilization of the task and $V_i = \sum_{j=1}^{i} U_i$ be the cumulative utilization of the first i tasks. We assume that the tasks are ordered so that $T_1 \leq \ldots \leq T_n$. We assume that a process will execute for a period of time q or until it

completes, whichever comes first. At the end of this period, the executive takes over and attends to interrupts, swaps in a ready process of higher priority if one is available, or continues the current process if it is of highest priority and not yet complete. The time for any of these activities is random depending upon the activity required. Indeed, the overhead may well depend on q with larger values of q creating larger overhead requirements. In this analysis, we assume that the overhead is independent of q and is given by a constant H. This analysis can be extended to allow H to depend on both q and n, the number of tasks being handled. This would require an exact formula describing the functional relationship. Future analysis will deal with such generalization.

A task is divided into quanta of processing. Each quanta generates an overhead period of H. This will result in an increase in processing time of CH/q and an increase in utilization of UH/q. The inflated total processor utilization is given by U(1 + H/q). We assume that the executive time required to handle I/O interrupts associated with this task is included in the processor requirement C. Indeed, the inflation factor (1 + H/q) is only approximate. C must be replaced C + f_c(C/q)H, where f_c(.) is the ceiling function. Consequently, we note

$$C(1 + H/q) \leq C + f_c(C/q)H \leq C(1 + H/q) + H.$$

This suggests using large q in order to minimize the inflation of the processing time of the task.

Using the quantum formulation, it is possible to analyze the worst case schedulability of the tasks. It is possible that the initialization time (and hence deadline) of a task can be controlled. It is, however, more likely that this cannot be controlled. Consequently, the initiation time of a high priority task can occur just at the beginning of the quantum of processing assigned to a lower priority task. The delay before the task begins processing can be as much as H + q. This would argue for a small value of q in order to minimize the starting delay to any process. Therefore, there is a trade-off to be made in the choice of q. This involves balancing the additional processing time arising from short quanta against the priority inversion that is likely to occur with long quanta. We seek to find an optimum quantum size. In a worst case formulation, we wish to find a task set having minimum utilization whose inflated processing requirement fully utilizes the processor; that is, if any of the processing requirements of any the tasks is increased, then some deadline is missed. We then wish to choose q to maximize this minimum utilization.

Consider first the case of a single task, n = 1. The task will be initiated at time 0, have deadline T and processing requirement C. In the worst case, an aperiodic task will have begun a quantum of processing just before 0 and will process for the length of a quantum. In order for the deadline to be

met, we must have $q + H + C + f_c(C/q)H \leq T$. Note that even though there is a single task, it must be executed in a quantum fashion and incur the extra overhead. Consequently, we approximate the situation by

$q + H + C(1 + H/q) \leq T$, or $U(1 + H/q) \leq 1 - (q + H)/T$, or $U \leq q/(q + H) - q/T$.

We wish to choose q in order to make the bound $q/(q + H) - q/T$ as large as possible. A straightforward calculation shows that the optimal q is given by $q = (HT)^{1/2} - H$, where we assume that $T > H$. The resulting bound on utilization is given by $U \leq (1 - (H/T)^{1/2})^2$.

If U satisfies this bound, then there is a range of choices of quanta which can be used, and we might pick the value of q which minimizes overhead. Thus for given H, T and U satisfying $U \leq (1 - (H/T)^{1/2})^2$, we might pick the *overhead minimization quantum size*,

$q = (T(1 - U) - H + ((T(1 - U) - H)^2 - 4UTH)^{1/2})/2$. Note we assume U, T and H are known and that the scheduling algorithm can use these quantities to establish the quantum size.

The above analysis applies to either a rate monotonic fixed priority algorithm or a deadline driven algorithm. We now extend to the multiple task case. The rate monotonic formulation is the easiest to carry out. Suppose that we are given n tasks and a critical instant. We consider subset of tasks consisting of the first i tasks, those having highest priority. Each task has its utilization inflated by the factor $1 + H/q$, and this subset may be delayed in processing $H + q$ units due to a lower priority task beginning processing just before the critical instant. In the Liu and Layland formulation, there is a bound on the worst case rate monotonic utilization given by $i(2^{1/i} - 1) = R_i$. In order to create the worst case full utilization of the critical zone, we can reduce the processing time of the i^{th} task by $H + q$ units. This means that the worst case will be given by $C_j = (T_{j+1} - T_j)q/(q + H)$, $1 \leq j \leq i - 1$ and $C_i = (2T_1 - T_i - (q + H))q/(q + H)$ where we now assume $C_i \geq 0$. This gives

$$V_i = [\sum_{j=1}^{i-1}(T_{j+1} - T_j)/T_j + (2T_1 - T_i - (q + H))/T_i]q/(q + H).$$

We wish to find the T's which minimize the above.

By repeating the Liu and Layland argument, we find

$V_i \leq R_i q/(q + H) \cdot q/T_i$, $1 \leq i \leq n$.

This says that for a quantum rate monotonic scheduling algorithm, the worst case bounds on a set of n tasks are given by

$\sum_{j=1}^{i} U_i = R_i q/(q + H) - q/T_i, 1 \leq i \leq n.$

This set of criteria illustrate the quantum trade-off. We wish to choose q to make $R_i q/(q + H) \cdot q/T_i$ as large as possible. Increasing q tends to make $R_i q/(q + H)$ near the Liu and Layland rate monotonic bound of R_i. Unfortunately, increasing q increasing the quantum loss factor q/T_i which serves to reduce the bound.

To better understand the situation, we rewrite the bound as $R_i X/(X+F) - X$, where $X = q/T_i$ and $F = H/T_i$. We assume that $X > F$. It follows that the optimal choice for X is given by $X = (R_i F)^{1/2} - F$. The resulting bound on V_i is given by $((R_i)^{1/2} - (F)^{1/2})^2$. It follows that all deadlines for the i^{th} class will be made if q is taken to be $T_i((R_i F)^{1/2} - F)$, and the cumulative utilization is taken to be no greater than $((R_i^{1/2} - (F)^{1/2})^2$. If all deadlines are to be met, then a quantum q must be found which simultaneously satisfies $V_i \leq R_i q/(q + H) \cdot q/T_i$. This cannot be pursued until a specific value for H and typical values for T_i are introduced. After this is done, one can determine specific bounds on processor schedulability. Furthermore, one should use an H which depends on the number of tasks being processed.

The quantum formulation for deadline scheduling is similar. We outline the mathematical details. The utilization of the task set, V_n, is inflated by the factor $(q + H)/q$, arising from the overhead. Furthermore, there is a potential priority inversion factor $q + H$ which arises when a higher priority task is held out during the course of a quantum of lower priority processing. This serves to reduce the period by $q + H$. As a consequence, we require in the worst case, $V_n \leq q/(q + H) - q/T_1$, where T_1 is the shortest period. One can optimize on q as before to find $q = (HT_1)^{1/2} - H$ and $V_n = (1 - (HT_1)^{1/2})^2$. This result is equivalent to rate monotonic result; however, R_i is replaced by 1.

The above analysis is somewhat approximate. We have assume that each task requires an integer number of quanta and that H is independent of the task set. Each of these assumptions can be modified. As we continue to study scheduling, we are able to introduce specific assumptions which will allow us to derive sharper bounds on processor schedulability. In addition, we mention below the possibility of a variable quantum formulation which could substantially increase the schedulability.

4 Variable Quantum Formulation

The previous section presented a worst case analysis for scheduling with hard deadlines when overhead is present using a quantum formulation. The bounds can be substantially below those of the no overhead case. The rigidity of the fixed quantum formulation can lead to serious inefficiencies

in scheduling. In this section, we introduce the idea of a variable length quantum. This approach seems to have much better scheduling power. A formal analysis will be carried out as a future task.

A variable quantum algorithm would operate in the following manner. For each task, the next initiation time, the next deadline, the task period and the amount of computation required would be kept in a database. At a decision point, the scheduling algorithm considers all ready tasks and will execute the one with the closest deadline. In addition, the scheduler will set a timer to indicate the time at which the next scheduling decision must be made. This time will be the end of the current procession requirement or at the initiation time of a task which is not yet ready but will become higher priority when it is ready. The current task is then allowed to execute until the timer signals a change is needed. This algorithm creates a deadline scheduler. It does generate some overhead, namely the overhead to carry out the computations themselves as well as the task swapping times. This overhead will, however, be considerably less than the overhead created by the fixed quantum scheduler. The great benefit occurs because a task need not be interrupted just to enforce a fixed quantum algorithm. Rather, any task can execute continuously if there are no required scheduling changes. For example, if we have a single task, then for the deadline to be met, we need $C+H \leq T$ or $U \leq 1-(H/T)$, a bound better than that given earlier for the fixed quantum formulation.

It may be desirable to limit the length of the processing times on a single task, for example to provide timely handling of queued interrupts. This and other generalizations are the subject of future work.

References

[Liu 73] Liu, C. L. and Layland, J. W.
Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment.
JACM , 1973.

4.7 Scheduling Hard Deadline Periodic Tasks with I/O

Scheduling Hard Deadline Periodic Tasks with I/O

E. Douglas Jensen, John P. Lehoczky

Lui Sha, Samuel Shipman and Martin McKendry

1. Introduction

1.1 Background

The current practice in designing systems for a real-time environment is rather *ad hoc*. Performance evaluation is accomplished either by simulation or by actual measurements on prototype systems, and system performance is improved by "fine tuning" certain system parameters. This hand-crafted scheduling approach often yields systems that are very difficult to understand, hard to maintain and hard to upgrade. Nevertheless, this approach has been taken in part because the theoretical foundation necessary to do better have not been established. The purpose of this work is to advance the understanding of task scheduling beyond the current uniprocessor stage to the consideration of network scheduling. It is intended to provide a foundation upon which high performance schedulers can be developed. This is an ambitious goal, especially for a distributed system. Such systems are composed of many processors, running many tasks distributed across those processors and are connected by a communication network. The scheduling involves meeting task timing requirements, and this entails scheduling the processors and the communication network. One must also integrate these two scheduling activities.

There is some literature on scheduling real-time tasks; however, all of those papers are narrowly focused on the scheduling of tasks in processors. None makes any mention of the importance of considering network scheduling. Consequently, these papers do not recognize that tasks do not run in isolation but receive data as input from other tasks (possibly over the network) and output data to other tasks (also possibly over the network). This suggests that we must begin with a broader view of the scheduling problem. Rather than focusing narrowly on the single programs resident in a processor, we must consider the input and output aspects of those tasks as well. This report documents our initial research on the network scheduling problem.

We begin with a short introduction to the literature on the processor scheduling of periodic processes with hard deadlines. This is followed by a description of the research project. We follow this with a description of the emulator and simulator software that was built as tools to facilitate our study of network scheduling. The source code for the simulator and the emulator is supplied in a separate binder.

We next present the results of this year's research and its implications for distributed system schedulers.

1.1.1 Scheduling Processors

There is some research literature on scheduling periodic tasks in processors. Liu and Layland[Liu73] considered the case of scheduling a single processor. Under the assumption that the deadline for a task is the same as the periodicity of the task, the optimal fixed priority and dynamic priority scheduling algorithms were determined. These optimal algorithms were shown to be the *rate monotonic* and *dynamic deadline* scheduling algorithms respectively. In this case, the word "optimal" means that if a task set can be scheduled by any fixed (dynamic) priority algorithm, then it can also be scheduled by the rate monotonic (dynamic deadline) scheduling algorithm. Liu and Layland derived the worst case scheduling bounds for these two algorithms. The worst case bound for the rate monotonic algorithm were shown to be ln 2 = 0.693. This means that any set of periodic tasks having processor utilization less than 0.693 can be scheduled by the rate monotonic algorithm so that no deadlines are missed. In fact, this algorithm assigns priorities in inverse order of the periods. The worst case bound for the dynamic deadline algorithm was found to be 1.0. This means that the dynamic deadline algorithm (which gives highest priority to the task having the nearest deadline) can schedule any set of periodic tasks which do not over-utilize the processor's capacity.

This work has been extended in a variety of ways. For example, Leung and Merrill [Leung 80] and Lawler [Lawler 81] considered deadline scheduling on multiple processors, while Martel [Martel 82] extended Lawler's work to introduce release times and due times. Mok [Mok 83] proved that the dynamic deadline scheduling algorithm remains optimal with respect to a mixed set of periodic and sporadic tasks with hard deadlines. A sporadic task has a random arrival time, a computation time and a hard deadline. Mok assumed a minimum separation time between two consecutive arrivals of sporadic tasks. Mok showed that the least slack time algorithm is also optimal. The slack time is the difference between the task deadline and the end of the task computation time.

1.1.2 I/O Scheduling

In contrast to the well rather developed theory of scheduling periodic tasks in processors, little if any work has appeared on the important problem of integrating the scheduling of tasks in processors with their data I/O. Given that distributed command and control systems are characterized by data flows from node to node with hard real-time constraints, it is imperative to develop a theory and to provide algorithms for hard real-time task scheduling with I/O.

The joint processor and I/O scheduling problem is especially difficult. In order to properly address it, we have taken a three pronged approach. This entails

- Continuing to extend the Liu and Layland theory to this more general setting,
- Developing a system emulator with which we can perform actual system experiments to

confirm the theory and observe the impact of effects which are typically ignored in analytic studies, and

- Developing a system simulator which will allow us to perform a large number of test runs against which to compare the emulator and the theory.

The three approaches offer three distinct but related ways to study the I/O scheduling problem. The analytic approach has the potential of giving general expressions for scheduling bounds over a wide range of cases; however, it necessarily entails making simplifying assumptions about the system. The simulator offers the possibility of a more realistic system model against which to compare the accuracy of the theory. The emulator offers the most realistic system view and by its nature incorporates system effects that are not included in the either the analytic models or the simulator. On the other hand, the results of the emulator may be influenced by the specific hardware and software characteristics. By having these three approaches, we can better evaluate the importance of specific system effects while producing theoretical results at the same time.

1.2 Overview of Progress

In this section, we briefly summarize the progress to date.

1.2.1 Progress of Theory Approach

It was mentioned in the introduction that the original basis for scheduling periodic tasks with hard deadlines in processors was developed by Liu and Layland [Liu 73]. This paper contains the results mentioned earlier on the optimality of and worst case bounds for the rate monotonic and dynamic deadline scheduling algorithms. In fact, this paper also contains two other results of even greater importance, because they form the entire basis for the worst case analysis. These results can be called the "hyperperiod result" and the "critical zone" result. The hyperperiod result offers the observation that for periodic processes, the pattern of initiation of each task is periodic. If one considers the least common multiple of all the periods (the hyperperiod), then the pattern of task initiations is repeated identically from one hyperperiod to the next. This leads one to the conclusion that one needs only to analyze the timing constraints over a single hyperperiod to determine if a particular algorithm can meet all the deadlines infinitely far in the future.

The "critical zone" result was used by Liu and Layland to determine if a fixed priority algorithm (specifically the rate monotonic algorithm) could meet all the deadlines. It indicates that to determine if a task will meet all its deadlines, one need only consider the first deadline of that task for the worse phasing of tasks. If that deadline is met when all tasks are initiated at a common time 0, then all deadlines of this task will be met.

It is important to note that both of these fundamental results no longer hold when one extends attention to the I/O scheduling case. The problem is especially acute when one allows for the postponement of deadlines, an important consideration which is realistic when one has additional buffers to store tasks. One of the initial findings was that one may have to consider many hyperperiods before a firm conclusion could be drawn about whether all deadlines would be met or some would be missed in the future. Furthermore, the ability of a task to meet the initial deadline did not allow one to conclude that it would meet all future deadlines, even in the static priority case when either I/O scheduling or deadline postponement is used. These important findings indicated that we needed to essentially abandon the Liu and Layland theory and develop a new one. The hyperperiod result was especially unsettling, because it had dramatic impact on the simulator and emulator. Simply stated, one generates a task set or the simulator or emulator, selects a scheduling algorithm and runs the tasks noting whether deadlines are made or missed. It is critical to know how long either must be run before a conclusion can be drawn. Clearly, the experiment can be stopped when a deadline is missed, but what if no deadline is missed? How long must one continue the run? It became important to settle this issue. Of special note was the development of a criterion to determine whether deadlines would be missed at some time in the future based on the behavior during an initial time segment. The theoretical research is discussed in greater detail in Chapter 2.

1.2.2 Progress on the Simulator

A simulator was built to allow for the study of factors usually ignored in analytic models and for the convenient study of a large number of cases. The simulator was designed to identify a threshold at which some task deadline would be missed. One begins with an initial task set defined in terms of the number of tasks, the period of each, and relative values for input DMA, processor computation and output DMA times. The simulator systematically scales up the utilizations until a task set is reached which cannot be scheduled by the particular algorithms being used. The simulator is able to handle all the scheduling algorithms described earlier. Furthermore, it allows for postponement of deadlines. The simulator also has a useful plotting and analysis facility to aid the user's understanding of the complex interactions that arise in I/O scheduling. The simulator is described in greater detail in Chapter 3.

1.2.3 Progress on the Emulator

An emulator was built to allow for the study of the behavior of the scheduling model described in Chapter 2 under realistic conditions in an actual computer system. It allows totally automated performance of sets of experiments, and provides complete traces of the significant events that occur during the experiment which, when analyzed by automated analysis tools, yield precise timings of the operations that determine the results of the scheduling algorithms in use. The structure of the

emulator allows different scheduling policies to be added easily. The emulator is described in detail in Chapter 4. Initial implementations of the rate monotonic and deadline scheduling algorithms exist. However, more extensive validation will be conducted next year. Some emulator results have already been used to demonstrate scheduling phenomena that occur in real systems, for example, priority inversion.

2. Theoretical Investigation

2.1 The Model

As we discussed in the introduction, this project utilizes a three pronged approach to the I/O scheduling problem. The accomplishments during this initial phase (April, 1985 - December, 1985) were of two types. First, we have made major developments in each of the three separate approach areas. These will be documented shortly. Second, we have made use of the software tools to run a pilot experiment on I/O scheduling. The conclusions drawn from this pilot study, while quite preliminary, are also quite clear. They lead us to see the vital importance of I/O scheduling. We will offer more specific conclusions shortly.

The initial phase of the research was devoted to the development of an appropriate model with which to consider I/O scheduling. This depends critically on the specific hardware architecture being considered. We selected a simple model which ignored a number of potential problems leaving enhancements to future efforts. The model has three parts:

1. Input DMA
2. Computation
3. Output DMA

We assume that a task has a period, and that the deadline is at the end of the period. This deadline can be extended by some integer number of periods if we consider the postponed deadline model. The postponement corresponds to the addition of buffers. In this year's work, we considered only two cases: no postponement and a postponement of one period. From the initiation time to the deadline, three things must be accomplished sequentially: the data associated with the task must be brought in by DMA, the computation must be carried out, and the data from the computation must be output by DMA. These three subtasks are treated as sequential and noninterfering activities. It is assumed that there is one input channel through which data can be brought into the processor and one channel through which data can be taken out. Furthermore these are assumed to be independent channels and do not steal any CPU cycles. This provides a good starting model and is a reasonable representation of some architectures. From the scheduling point of view, there are three places where scheduling can take place: One can schedule the input DMA phase, giving priority to whatever transfer is ready to be carried out, and one can similarly schedule the output DMA phase. Finally, there is a processor scheduling phase in which the processor can give priority to any task which is currently ready to run.

2.1.1 I/O Scheduling Model

In this study, we assume that each periodic task, T_i, has a fixed period P_i. Associated with each period, there are data needed input for the computation, I_i, the computation, C_i, and the data output from the computation, O_i. In other words, the scheduling of a task includes three phases: to schedule the DMA process which transfers the data from an input channel to the main memory, to schedule the CPU to perform the computation and to schedule the output DMA process which transfers the results of a computation to an output device. The deadline of a task initiated at period i can be designated as the beginning of next period, (i + 1). In this case, within the period, (i, i + 1), a task must complete all three phases of activities: I_i, C_i and O_i. This type of deadline designation represent certain demanding tasks where delays must be minimized as much as possible. On the other hand, the deadline of a task initiated at period i can be postponed to the beginning of period (i + 2) or even later. By postponing deadlines, it becomes possible to input the data at period i and perform the computation and data output at period (i + 1) or later. Deadline postponement improves the schedulability at the cost of increased buffer requirement and phase delay. If we postpone the deadline to infinity, then the scheduling problem becomes a three stage queueing network with infinite buffers space.

In this study, we assume that for each task T_i, the period P_i, the input data I_i, the computation C_i and the output data O_i are deterministic. In addition, we assume that the effect of "cycle stealing" during DMA is negligible and there is no contention between input channels and output channels. In other words, we assume that we have high performance computers which have dual-port memory banks and separate input and output channel controllers. These assumptions will be relaxed in the next year's study.

2.1.2 Scheduling Algorithms

We next discuss the selection of scheduling algorithms. There are fundamentally two major classes of algorithms and subclasses within. The two major classes are

1. The algorithms which treat each of the three phases of scheduling independently, and
2. The algorithms which integrate all three scheduling activities together.

2.1.2.1 Nonintegrated Algorithms

The first class of algorithms essentially generates a separate scheduling algorithm for each of the three phases. We have chosen to consider three possible choices for each of these single phase schedulers:

1. FIFO,
2. Rate monotonic, and

3. Deadline.

The FIFO algorithm is a model of no scheduling algorithm at all. It merely schedules the tasks in the arrival order. This algorithm will be seen to be very unstable. It is possible that the tasks will arrive in the order which is consistent with a good scheduling algorithm and so be scheduled correctly by FIFO. It is equally possible that the tasks will arrive in an order which will lead to poor scheduling decisions. Consequently, we will see that FIFO will have a wide range of performance, but is typically inferior to an algorithm which is designed to meet deadlines at high utilization levels.

The rate monotonic algorithm was shown by Liu and Layland to be the optimal fixed priority algorithm. It is very easy to implement in the processor, and can be implemented for input and output DMA processes with some modifications to the channel controllers. Specifically, the scheduler gives higher priority to task i over task j if task i has a shorter period that task j. We will see that its worst case utilization bound (0.693) is a large underestimate of its typical potential.

The dynamic deadline algorithm was shown by Liu and Layland to be the optimal dynamic priority algorithm. Specifically this algorithm gives the highest priority to the task having the nearest deadline. This can be separately implemented at each of the three stages of scheduling.

2.1.2.2 Integrated Algorithms

This study included one algorithm from the second group consisting of the algorithms which integrate all three stages of scheduling. We considered the *propagated deadline algorithm*. This algorithm works in the following way. Suppose a task T_1 has period P_1 and requires I_1 units of DMA input time, C_1 units of processor computation time, and O_1 units of DMA output time. Supposed that this task is initiated at time $t = 0$. At the first period, the propagated deadline algorithm treats this task as having a deadline given by $P_1 - O_1 - C_1$ for purposes of DMA input scheduling, a deadline of $P_1 - O_1$ for purposes of processor scheduling , and a deadline P_1 for purposes of DMA output scheduling. This algorithm is similar to the dynamic deadline algorithm applied at each stage, but it takes later computation requirements into account in earlier scheduling decisions. This algorithm is an improvement over three independent scheduling algorithms, but we will later see that the increase in complexity may not be worth the slight increase in scheduling potential.

2.2 Methodology

2.2.1 Algorithm Selection

We briefly discussed the selection of candidate algorithms in Chapter 2. There, the distinction was made between algorithms which treated each of the three phases of scheduling in isolation from the others and algorithms which used an integrated view. We chose four representatives from the first category (FFF, FRF, RRR, and DDD) and one (PPP) from the second category. Within the first category, the four were chosen to give representation to algorithms which exhibit a lack of a systematic use of scheduling theory and to those which make use of concepts of real-time scheduling. The latter consist only of simple and effective linear algorithms, since complex algorithms are self-defeating in the context of hard real-time scheduling.

In selecting an unsophisticated algorithm, we need a model for the practice of I/O scheduling commonly seem in many systems. In most existing channel processors, the priority of a channel, once assigned, remains unchanged during run-time. In addition, all data I/O to or from a channel contends at the channel's priority, independent of the nature of the tasks. This practice can create great difficulties for distributed systems. For example, the system database is shared by many parts of the various subsystems. However, all the data provided by the database system must contend at the same priority assigned to the disk channel, independent of the urgency of the requests. The general practice of assigning channel processor priorities is to do it according to the average speed of the devices it connects. In addition, once a channel starts a DMA operation, it is not preemptable. This type of scheduling policy is approximated by the FIFO scheduling algorithm.

Since most existing computers are able to support the rate monotonic scheduling algorithm for CPU scheduling, the set of algorithms: FIFO for input DMA, Rate Monotonic for CPU and FIFO for output DMA (FRF algorithm) is designed to measure the effect of using good CPU scheduling algorithm but a poor I/O scheduling algorithms. The three stage FIFO (FFF) is selected to model the effect of a complete lack of sound real-time scheduling algorithms. Since the rate monotonic scheduling algorithm and the deadline algorithm represent the optimal static and dynamic scheduling algorithms for single stage scheduling, it is logical to use them at each stage of scheduling. Thus, two candidates for task scheduling with I/O are the three stage Rate Monotonic algorithm (RRR) and the three stage Dynamic Deadline scheduling algorithm (DDD).

In this study, we introduce a refined version of the earliest deadline scheduling algorithm called the *propagated deadline* scheduling algorithm. This algorithm was described in detail earlier and is designed to provide an integrated approach to scheduling.

2.2.2 The Halting Theorem

In either the simulator or the emulator, a very important problem is the halting problem. That is, when a deadline is missed for a particular task set, it is a certain event. However, when no deadline is missed by a given time in the simulation/emulation process, it is unclear if all the deadlines will continue to be met in the future. In a pure processor scheduling formulation, one needs to trace no more than a single hyperperiod (the least common multiples of all the periods). However, this result no longer holds when either I/O scheduling or deadline postponement is introduced. Consider the case of pure processor scheduling with a single deadline postponement. Suppose that tasks T_1 and T_2 have periods 5 and 15 and that the processor utilization of the two tasks are 1/5 and 13/15 respectively. In addition, let the scheduling algorithm be the rate monotonic scheduling algorithm. In this case, the hyperperiod is 15 and the total utilization is 16/15 > 1. It is obvious as the deadline will be missed, because there is 1 unit of extra work for each hyperperiod. However, no deadline is missed until 15 hyperperiods have passed. Since the number of hyperperiods needed to detect a missed deadline could be arbitrary large, it is important to determine criteria to determine whether all deadlines will be met without carrying on the simulation/emulation process indefinitely into the future.

Given that no deadline has been missed, a set of sufficient conditions for stopping the simulation/emulation is the following theorem:

The Halting Theorem: The deadlines for a given set of periodic tasks under a given scheduling algorithm will always be met, if

- No deadline is missed within the first two hyperperiods;
- The total amount of input DMA (computation, output DMA) carrying over from the first hyperperiod into the second hyperperiod is less than or equal to that from the second hyperperiod into the third hyperperiod.
- The first idle time slot for input DMA(computation, output DMA) in the third hyperperiod is, if it exists at all, no later than that in the second hyperperiod.

We outline the reasoning which makes these sufficient conditions. The first condition is obvious. The second condition states that work must not keep piling up from one hyperperiod to the next hyperperiod for either input DMA, computation or output DMA. The third condition requires some explanation. Starting from the beginning of each hyperperiod, the arrival of tasks is identical owing to the periodicity of tasks. However, the carryover from one hyperperiod to the next may not have the same effect even if the total units of work is the same for each of the three phases. This is because tasks can be of different priority. However, all the carryover from a previous hyperperiod can be treated as initial conditions for a given hyperperiod. Beginning with the first idle slot of each schedul-

ing phase, the effect of the carryovers at that phase vanishes. For example, suppose that the first idle slots for input DMA, computation and output DMA exist and are 17, 21, 57 for both the second and third hyperperiods. In addition, the total amount of carryover for each of the three phases are 2, 7, 11 for both the second and third hyperperiods. In this case, we know that all the deadlines will always be met as long as there is no deadline missed within the first three hyperperiods. The reasons are twofold. First, the carryover does not increase from one hyperperiod to the next hyperperiod. Second, the impact of carryovers is not made worse from one hyperperiod to the next hyperperiod.

Given that a criterion for halting the simulation or emulation has been created, one can now consider a pilot study to provide a preliminary evaluation of the various scheduling approaches. This pilot study was fully described in an earlier section.

3. The Simulator

A simulator was built to allow for the convenient study of a large number of cases. This tool was specially designed to take a basic task set and continue to increase the utilization until a breaking point was reached. One begins with an initial task set defined in terms of number of tasks, period for each, relative input DMA, CPU, and output DMA utilizations. The program systematically searched for that task set which was schedulable, but for which any increase in utilization led to an unschedulable situation. The simulator allowed for a wide variety of combination of scheduling algorithms and for the postponement of deadlines. It also had a useful plotting and analysis facility to aid the user's understanding of the complex scheduling problem.

3.1 Users' Guide

The simulator was written in standard Pascal. To run the program, a user only needs to answer the prompts. There are 3 sets of options that a user can select. The first set concerns the selection of scheduling policies. If a user chooses the auto-mode, then the simulator will run each task set with 5 sets of policies: the 3 stages FIFO, the 3 stage Rate Monotonic, the 3 stage Deadline, the 3 stage Propagated Deadline and finally the set of algorithms consisting of FIFO for input DMA scheduling, Rate Monotonic for CPU scheduling and FIFO for output DMA scheduling. The last set of algorithms is used to model the effect of having a good processor scheduler but no good I/O scheduler. Instead of using the auto-mode, a user can choose the manual mode to specify scheduling algorithms for each stage of scheduling. There are four algorithms to choose from: FIFO, Rate Monotonic, Deadline and Propagated Deadline. Finally, in either auto-mode or manual mode, users will be asked if they want to postpone the deadline.

Having specified the scheduling algorithms, one needs to specify a set of tasks. The specification has two modes: the statistical mode or the manual mode. If the statistical mode is chosen, a user will be asked to specify the number of tasks, the average ratio between input DMA, computation and output DMA and the average phase delay of tasks' starting times. Once these parameters are given, the task set will be generated randomly. If the manual mode is chosen, then a user must specify each task in turn by answering all the prompts for each task.

After specifying both the scheduling algorithms and tasks, a user will be asked to select the option to control the execution and style of reporting. First, a user will be asked if he wishes to automatically search for the break down threshold for the given task set pattern for each of the scheduling policies. If the answer is "yes", the threshold will be reported. Otherwise, the task set will be executed and the results of execution will be reported. The reporting style can be either verbose or terse. In the

verbose mode, each task is described in detail, and a user has the option to plot the scheduling process. In the non-verbose mode, only a summary of the execution results will be given. Finally, the results of execution will be recorded in a output file called Result.sch. There is another output file called, EMU., which is used to drive the emulator to obtain a cross validation. For the use of the file EMU., please refer to the chapter on the emulator. Both of the files will be rewritten when the simulator program is executed again. Therefore, the results should be copied to another file before reexecute the simulation program. In the following, we give two examples to illustrate the use of the simulator.

3.2 Examples

In this section, we give two examples to illustrate the use of this simulator. Example 1 uses the manual modes and plots the scheduling process. Note that in the plotting, the word "end" marks the end of a hyperperiod. Example 2 uses the auto-modes to have a comparative study of different scheduling algorithms. Note that for each set of scheduling algorithms, the loads at which all the deadlines are just met and the loads at which deadlines are just missed are reported. Because of the integer nature of the periods and computation times, these two values are sometimes not very close.

3.2.1 Example 1: The Manual Approach

```
[LNKXCT SCHEDU Execution]
The results of this execution are recorded in the  file result.sch.
To save the record, copy it before executing this program again.

Scheduling policy can be specified manually for each stage.

In auto-mode, the program steps through
1:FIFO, 2:Rate Monotonic, 3:Deadline,
4:Propagated deadline, 5:FIFO I/O and Rate Monotonic CPU. Each policy set
can be  run with 0 and 1 postponed deadline.

The style of execution can be either verbose or not.

policy mode: auto, manual (a/m)m

Scheduling Policy Selection Guide
1: FIFO, 2: Rate Monotonic, 3: Deadline, 4: Propagated Deadline
Specify the No.  1 set of scheduling policies
DMA-in scheduling policy = (1..4)1
CPU scheduling policy = (1..4)1
DMA-out scheduling policy = (1..4)1
Deadline Postpone = (0..1)0
Specify a new set of policies (y/n)n
generate the task set randomly (y/n)n
```

```
input task   1 parameters
period =(  3< P <  700)20
the hyperperiod now  is           20

DMAin time =5
Total DMA-in Load now is    0.2500

Computation time =5
Total Computation Load Now is    0.2500

DMAout time =5
Total DMA-out load now is    0.2500

The initial starting time of the task=(>=0)0
specify next task (y/n)y

input task   2 parameters
period =(  3< P <  700)40
the hyperperiod now  is           40

DMAin time =8
Total DMA-in Load now is    0.4500

Computation time =8
Total Computation Load Now is    0.4500

DMAout time =8
Total DMA-out load now is    0.4500

The initial starting time of the task=(>=0)0
specify next task (y/n)n

Automatically search for the scheduling thresholds (y/n)n
Verbose Mode (y/n)y


     Task     Period     DMAin       CPU    DMAout     Phase
        1         20         5         5         5         0
        2         40         8         8         8         0

  2 tasks              0.4500    0.4500    0.4500        0%

FIFO             FIFO             FIFO             Deadline postpone =    0


All deadlines are met within  2 hyperperiods. No Carryovers.
plot the timelines (y/n)y
(1: From beginning, 2: Around the end of Hyperperiod, 3: both)1
(late execution  indicated by negative Number)

 DMA in:  1  1  1  1  1  2  2  2  2  2  2  2  2
 CPU   :                 1  1  1  1  1           2  2  2  2  2  2  2
 DMAout:                                 1  1  1  1  1
```

```
      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

DMA in:  1  1  1  1  1
CPU   :  2                1  1  1  1  1
DMAout:     2  2  2  2  2  2  2  2     1  1  1  1  1
      0 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39end

DMA in:  1  1  1  1  1  2  2  2  2  2  2  2  2
CPU   :                   1  1  1  1  1           2  2  2  2  2  2  2
DMAout:                                    1  1  1  1  1
      0 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60

DMA in:  1  1  1  1  1
CPU   :  2                1  1  1  1  1
DMAout:     2  2  2  2  2  2  2  2     1  1  1  1  1
      0 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79end

DMA in:  1  1  1  1  1  2  2  2  2  2  2  2  2
CPU   :                   1  1  1  1  1           2  2  2  2  2  2  2
DMAout:                                    1  1  1  1  1
      0 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99100

Continue the plot (y/n)n


continue = (y/n)n

EXIT
```

3.2.2 Example 2: The Automode Approach

```
[LNKXCT SCHEDU Execution]
The results of this execution are recorded in the  file result.sch.
To save the record, copy it before executing this program again.

Scheduling policy can be specified manually for each stage.

In automode, the program steps through 1:FIFO, 2:Rate Monotonic, 3:Deadline,
4:Propagated deadline, 5:FIFO I/O and Rate Monotonic CPU. Each policy set
can be  run with 0 and 1 postponed deadline.

The style of execution can be either verbose or not.

policy mode: auto, manual (a/m)a
Run postponed deadline (y/n)n
generate the task set randomly (y/n)y

no of tasks (<=  30) 10

Average ratio for DMAin, Computation and DMAout (e.g. 2 4 1)   7   5   7
```

```
phase delays: auto(Mean=0 and 50% of period), manual(a/m)m

specify average phase delay as X% of period, X=  0

Automatically search for the scheduling thresholds (y/n)y
Verbose Mode (y/n)n
- - - - - - - - - - - - - - - - - - - -

     Task     Period     DMAin       CPU    DMAout     Phase

 10 tasks              0.4452    0.3039    0.3928        0%

FIFO             FIFO              FIFO              Deadline postpone =    0

All deadlines are met within  2 hyperperiods. No Carryovers.

     Task     Period     DMAin       CPU    DMAout     Phase

 10 tasks              0.4898    0.3039    0.3984        0%

FIFO             FIFO              FIFO              Deadline postpone =    0

Task:  6, instance:  1, first missed deadline at time     37
- - - - - - - - - - - - - - - - - - - -

     Task     Period     DMAin       CPU    DMAout     Phase

 10 tasks              0.8904    0.6079    0.7857        0%

Rate Monotonic   Rate Monotonic   Rate Monotonic   Deadline postpone =    0

All deadlines are met within  2 hyperperiods. No Carryovers.

     Task     Period     DMAin       CPU    DMAout     Phase

 10 tasks              0.9349    0.6079    0.7912        0%

Rate Monotonic   Rate Monotonic   Rate Monotonic   Deadline postpone =    0

Task:  6, instance:  1, first missed deadline at time     37
- - - - - - - - - - - - - - - - - - - -

     Task     Period     DMAin       CPU    DMAout     Phase
```

```
 10 tasks            0.8904   0.6079   0.7857       0%

Dead Line       Dead Line       Dead Line       Deadline postpone =   0

All deadlines are met within  2 hyperperiods. No Carryovers.

     Task    Period    DMAin      CPU   DMAout    Phase

 10 tasks            0.9349   0.6079   0.7912       0%

Dead Line       Dead Line       Dead Line       Deadline postpone =   0

Task:  6, instance:  1, first missed deadline at time    37
- - - - - - - - - - - - - - - - - - -

     Task    Period    DMAin      CPU   DMAout    Phase

 10 tasks            0.9484   0.6484   0.8293       0%

PPG Dead Line   PPG Dead Line   PPG Dead Line   Deadline postpone =   0

All deadlines are met within  2 hyperperiods. No Carryovers.

     Task    Period    DMAin      CPU   DMAout    Phase

 10 tasks            0.9984   0.6706   0.8793       0%

PPG Dead Line   PPG Dead Line   PPG Dead Line   Deadline postpone =   0

Task:  1, instance: 18, first missed deadline at time  1261
- - - - - - - - - - - - - - - - - - -

     Task    Period    DMAin      CPU   DMAout    Phase

 10 tasks            0.4984   0.3039   0.4341       0%

FIFO            Rate Monotonic  FIFO            Deadline postpone =   0

All deadlines are met within  2 hyperperiods. No Carryovers.

     Task    Period    DMAin      CPU   DMAout    Phase

 10 tasks            0.4984   0.3317   0.4341       0%
```

```
FIFO            Rate Monotonic  FIFO            Deadline postpone =   0

Task:  6, instance:  1, first missed deadline at time    37
continue = (y/n)n

EXIT
```

4. The Emulator

The scheduling emulator is a computer system designed to serve as a testbed for experimentally investigating the applicability of the scheduling theory we have developed and will develop in the course of this ongoing work. In the arena of computer systems, theoretical results often are not considered relevant until their applicability has been demonstrated in a real, working system. We have reached the stage in this investigation that such a demonstration of the theoretical principles heretofore derived is called for. It is entirely fair to expect experimental validation of these results, because a real system will usually be subject to constraints that violate certain assumptions of the theoretical models and the extent of these effects must be determined.

Our major requirements for the emulator are that it:

- support a realistic model of scheduling that corresponds as closely as possible to the model specified in Chapter 2;
- support the scheduling algorithms we plan to study and facilitate the addition of any other scheduling algorithms that may become interesting to us;
- be modular and easy to modify along other dimensions as well;
- allow observation of its internal behavior at a fine granularity, and allow highly precise timing measurements to be performed;
- be relatively easy to use;
- require a reasonable amount of effort to build in the time available (since no existing system that we know of satisfies the foregoing requirements).

The emulator meets all its goals satisfactorily.

It is interesting to contrast the roles of the simulator and emulator in our research. The theoretical model of scheduling described in Chapter 2 is already sufficiently complicated that experimentation is needed to understand its behavior. Working out examples by hand becomes cumbersome and time-consuming—an automated tool is very useful, and the simulator serves this need. The simulator is better for this kind of work, because the emulator takes much longer to process a single test, is less oriented toward interactive work, and produces a large amount of data for each test.

For the same reason, the simulator is better able to perform the long searches required to find the scheduling breakdown points. However, the emulator can provide valuable details about what happens in a real system when the tests that are near the scheduling breakdown point are run. Also, results from the emulator help to validate the simulation results (and vice versa). In short, the roles of the simulator and emulator complementary, and both are practically necessary for this kind of investigation. The following sections provide a more detailed description of the emulator.

4.1 Overview

This section provides an overview of the emulator's structure and how it operates. Details of its implementation are discussed in Section 4.2; details of its operation are given in Section 4.3.

The requirements for the emulator listed above led to its design in a straightforward way. The scheduling model states that the actual *processing* of input DMA, task execution, and output DMA, goes on without interference among the phases. This implies that they can be performed sequentially for purposes of emulation. The results of the input DMA scheduling determine the task initiation times for the processing phase, and the completion times of tasks determined by the processing phase determine the initiation times for ouput DMA. The results of output DMA scheduling reveal whether or not deadlines have been met. Therefore, the emulator consists of three sequential phases: input DMA, processing, and output DMA.

In this year's work, the processor is the focus of realism in the emulator. Once processor scheduling under realistic conditions is better understood, attention can be shifted to the details of I/O devices, applying the lessons learned from applications processor scheduling.

So, very generally, the emulation proceeds in three steps:

pre-processing	From a description of the task set, appropriate input data for the emulator is generated, and the input DMA scheduling is simulated to provide a list of task initiation times for the emulation.
execution	The emulator is executed, using the list of task initiations produced in the preceding step to drive the emulation. This produces a record of the task completion times for each task initiation.
post-processing	The task completion times are input to a simulation of the output DMA scheduling, and the final output completion times are compared to the task deadlines for each task initiation, to determine whether or not deadlines were met.

The task descriptions include the relevant task parameters described in Chapter 2:

1. period
2. time required for input DMA
3. time required for task processing
4. time required for output DMA
5. phase (time of first initiation)
6. time required for input interrupt service

The pre- and post-processing phases involve using various software tools developed for the emulator running on the development machine under the Unix operating system. These programs

which are described in some detail in the sections on operation and internals. The execution phase is more interesting, since it must emulate a real, stand-alone system.

The emulator execution phase uses a CPU that runs an operating system kernel, which manages the execution of applications tasks. So far, this sounds like any other typical computer system. It is called an emulator because:

1. The "applications" tasks do not accomplish any real application; they just use up the amount of CPU time specified for them in the task descriptions.

2. I/O is handled differently than in a real system.

In real systems, and in the emulator, an input interrupt signals the completion of an input DMA operation to the processor, and results in an application task being dispatched in response to the input. However, there are not any real I/O devices in the emulator (since no real I/O is going on), and tasks are only dispatched in response to input interrupts. Therefore, there must be something to cause the input interrupts which drive the system. Also, we know that the input interrupts must occur according to the schedule produced by the pre-processing phase of the emulator. So we need some kind of device that will produce the interrupts at the specified times. This can't execute on the applications processor (perhaps "simulating" the occurrence of interrupts), since it would interfere with the measurements by introducing an overhead that would not be present in a real system.

To understand our approach to this problem, consider the hardware configuration shown in Figure 4-1. This is a standard node of the Archons Interim Testbed system, which also supports the Archons project's Alpha distributed operating system kernel. Each node consists of a Multibus chassis containing the following:

- two 68010-based Sun Microsystems Version 1.5 processor boards (with 1M byte of on-board memory each).
- one 0.5M byte Multibus memory board
- one 3Com Ethernet controller (with Ethernet tap)
- one custom-built interrupt generator board (to provide inter-processor interrupts)

With this in mind, the emulator execution phase is implemented as follows.

One of the processors runs VHS, a simple kernel that provides multi-tasking and scheduling to run the applications tasks. This kernel is not radically different from most small kernels, except that it is smaller than most, since any functionality not directly relevant to scheduling was omitted. VHS will be discussed in more detail in Section 4.2. In the diagram, this processor is designated "VHS Host."

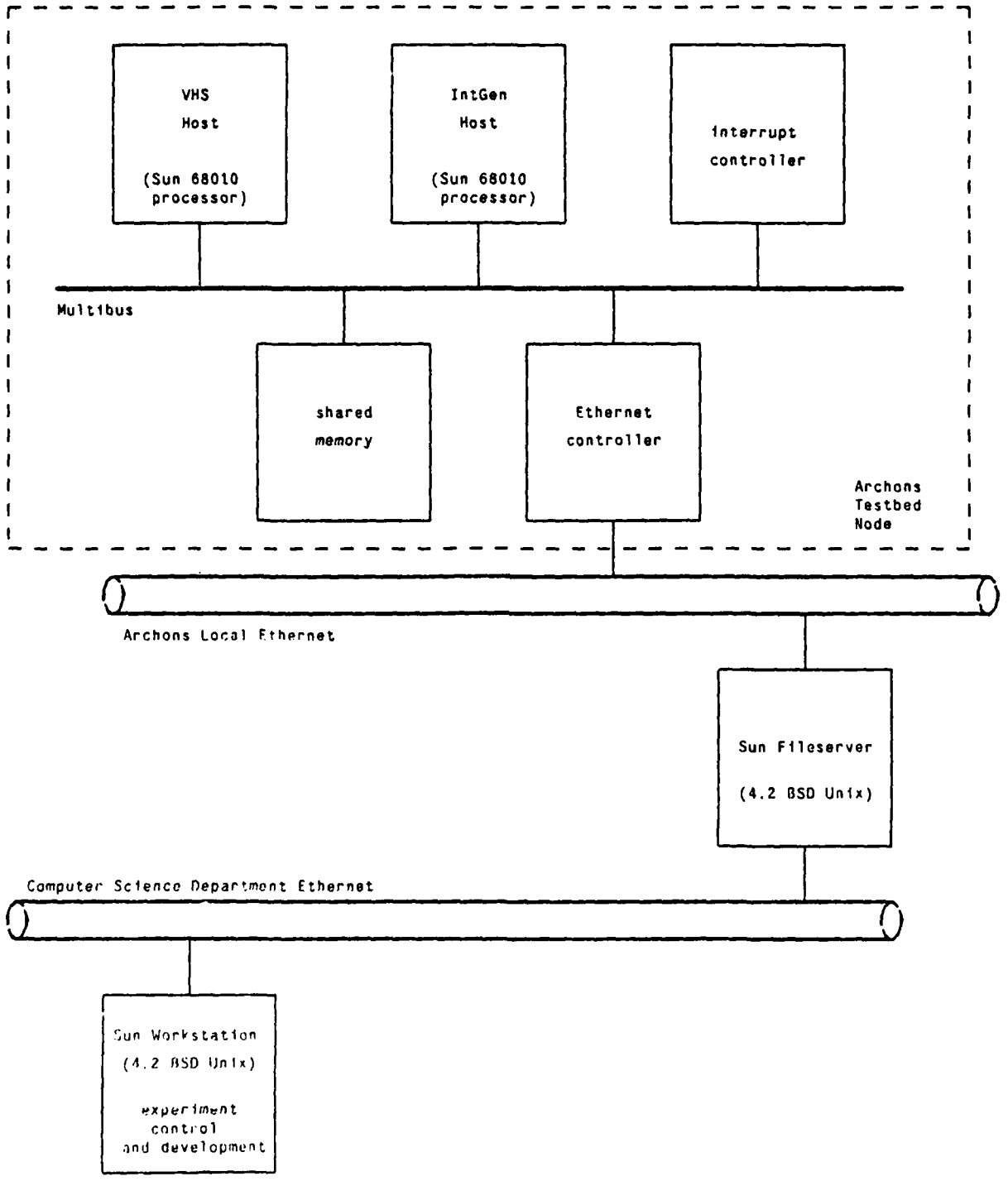


Figure 4-1: Archons Interim Testbed Node

The other processor, designated "IntGen host," runs a simple program called IntGen ("Interrupt Generator") that causes the I/O interrupts to be generated at the intervals specified in the list of task initiation times (hereinafter referred to as the "interrupt log") produced in the pre-processing step. The interrupt log is downloaded with the IntGen program, which uses the Sun 1.5 processor's timer facilities to time the intervals between interrupts. Since the Sun 1.5 processors can't directly cause interrupts on the Multibus, the IntGen host uses the interrupt generator board to generate them. This board is a simple Multibus device that will cause an interrupt of a specified Interrupt Priority Level (IPL) when a value is written to one of its memory-mapped I/O registers. (It was custom designed and built by another member of the Archons project, J. Duane Northcutt, in support of this effort and the Alpha kernel effort.)

In short, the IntGen program runs on one processor and drives the emulation by providing input interrupts to the VHS processor at the intervals determined in the pre-processing phase. VHS runs and schedules the tasks specified in the task description, using the specified scheduling algorithm, in response to the input interrupts produced by IntGen. This arrangement is illustrated schematically in Figure 4-2.

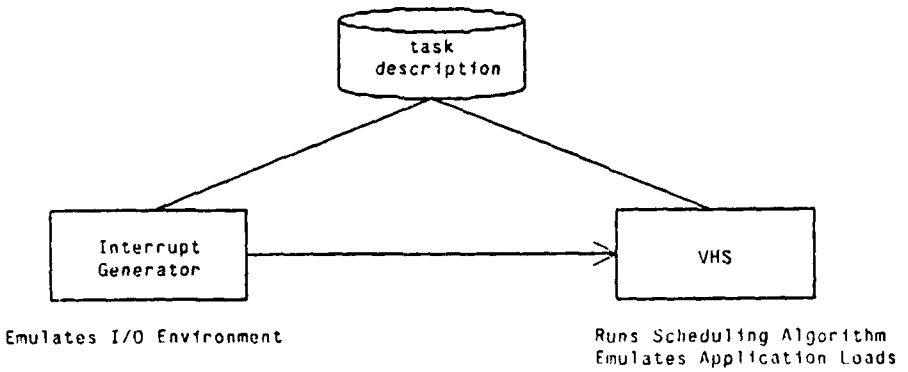


Figure 4-2: Emulator System Organization

The Ethernet controller is used only for downloading IntGen and VHS, and for uploading the results of the emulation. The Multibus memory is currently unused, but may be used to hold the interrupt log, should it ever grow too large to be kept in the processors' local memories.

For DMA scheduling, the emulator currently supports the following algorithms, in preemptive and non-preemptive versions: FIFO, rate monotonic, deadline, and propagated deadline. For processor scheduling, the following algorithms are supported in pre-emptive form only: rate monotonic, deadline, propagated deadline. Also, the emulator supports postponed deadlines (as described in Chapter 2).

4.2 Emulator Internals

This section discusses the internals of the emulator, and serves as a combination of design rationale, design description, and implementation description. It provides fewer details than separate documents would, and attempts to focus on those details as they relate to the design requirements and the nature of the scheduling problems we are studying. We discuss the VHS kernel first, then, in less detail, the other tools required for emulation.

4.2.1 VHS Internals

We will approach VHS internals by first explaining how our requirements for the emulator led to certain design goals. Then we give a brief, general description of VHS. Next we go into more detail on the more important parts of the kernel, and finally discuss sources of variation in measurements relating to emulation results.

4.2.1.1 Goals and Description

The requirements for the emulator (listed on page 19) seem to us to imply two major design goals:

- observability of internal state
- powerful underlying mechanisms

Observability of system state is an important requirement, since we have to be able to find out fine-grained details of what happens during the execution of tests to answer any questions that might arise about the behavior of the scheduling algorithms and effects of factors that arise in emulation but not in simulation. However, it is not required that these details be known in real time--in fact, since task execution times are measured in tens of milliseconds, that would be impractical. We chose to keep a log of significant events in the kernel as they occur, and examine the log after the experiment is over to find out what happened. The mechanisms we use to do this are discussed in subsequent sections.

Powerful underlying mechanisms are important in the VHS design. Each of the major facilities provided by the VHS kernel is supported by its own particular abstraction, as we will see. The major facilities are task control, scheduling, and data gathering, and are discussed in more detail below.

4.2.1.2 Details

Some basic decisions about implementation were made at the outset. With the software development tools and expertise at our disposal, it was clear that the best approach was to write the kernel in the C language, using as little assembly code as possible for the lowest level machine dependent parts (such as cold-start and context switching). The emulator was developed on a Sun-2/120

workstation running the Sun 1.3 release of the Berkeley 4.2BSD Unix operating system. Both VHS and IntGen are standalone programs that run on the Sun 1.5 processor. A listing of the source code is provided as an appendix to this report, but the following discussion does not make any direct reference to the code.

The VHS kernel and all user tasks share the same address space, and run in supervisor mode. Task are created statically, only at system-build time, and are never destroyed. Elaborate inter-task protection schemes and dynamic allocation and deallocation of resources were considered irrelevant to the purpose of the emulator, and therefore were not implemented. However, nothing in the system precludes adding any of these features, should they ever become necessary.

Task control is the first major facility we will examine. It is achieved through the use of a standard low-level synchronization mechanism, the counting semaphore. As we mentioned earlier, each application task is specified statically at system build time, and is created and made ready to run at run time when the system is initialized. Each task has the following structure:

```
begin TaskN
    do forever
        wait on TaskN's semaphore
        use CPU time specified for TaskN execution
    end do
end TaskN
```

When an input interrupt occurs, the following is done:

```
begin IOInterruptN
    clear interrupt condtion
    compute task deadline
    signal TaskN's semaphore
end IOInterruptN
```

Each task automatically finishes processing for each initiation before starting processing for the next (or pending) initiation. In other words, if a task is processing on behalf of input interrupt *n* and input interrupt *n + 1* occurs, the task finishes its processing of interrupt *n* before starting to process interrupt *n + 1*. The count in the counting semaphore keeps track of how many initiations are pending for a particular task (few, we hope, if the tasks are to be schedulable).

The semaphore is a simple, standard synchronization mechanism that forms the basis of the task control system described here. It is sufficiently general that we are confident it will remain useful if the system is enhanced. Possible enhancements could include consideration of precedence relationships among tasks or synchronization activities (such as mutual exclusion of access to resources) among tasks that create dependencies that must be accounted for in scheduling. These could be controlled using semaphores as well.

The next important facility is scheduling. In VHS, we do not attempt to provide complete

policy/mechanism separation in the scheduler, but provide a mechanism to allow different scheduling policies, possibly supported by different underlying mechanisms, to coexist in the system and to be selectable at system startup. This is sufficiently flexible for our purposes, and we consider it a good compromise between a fully general policy-based scheduler, which is very difficult to build and would involve unacceptable amounts of overhead, and an inflexible scheduler, which what is usually provided by operating systems today, but clearly useless for our purposes.

Each policy must be implemented by providing a set of routines that correspond to a set of standard scheduling interface routines recognized by the kernel. In some cases, modifications may have to be made to other parts of the system to provide the information needed for the scheduling policy to operate (such as priorities). The set of standard scheduling interface routines are described below. Note that each task has a task control block (TCB) that contains all the system's knowledge about that task, and a scheduling control block (SCB) that contains the scheduler's knowledge about that task (and is part of the TCB). Also, each task has a task identifier (TID) unique to that task.

TaskInit()	Initializes data structures used by the scheduler, and sets up the idle task, which runs when no other task is ready to run.
Init(TCB, SCB)	Initializes the scheduling control block for the task corresponding to TCB with values from SCB.
Ready(TID)	Makes the task whose identifier is TID ready to run (but does not reschedule—when the next reschedule occurs, the change in status takes effect).
Block(TID)	Blocks the current task (but does not reschedule—when the next reschedule occurs, the change in status takes effect).
Resched()	Will reschedule if necessary, depending on the scheduling policy. If it does reschedule, it determines which task should be running. If it is the current task, it does nothing. If it is a different task, it changes the current task state to ready to run (unless it is already blocked), and switches to the new task.

By convention, these routines are named Sp*n*TaskInit, Sp*n*Init, etc., where *n* is the scheduling policy number (e.g., Sp1Block, etc.). The entry points corresponding to these routines for each scheduling policy are entered in a jump table indexed by the scheduling policy number (determined at system startup). In the code, a macro that indirects through the jump table is used to call these routines, so the "generic" routine call READY() (for example) accesses the actual routine Sp2Ready if scheduling policy 2 is being used. Each scheduling policy is a separate code module in the kernel, by convention. The existing modules schpol1.c and schpol2.c provide examples.

Although this method allows different mechanisms for different policies, the current set of policies

uses the same underlying abstraction—a queue abstraction, implemented by a queue management package (in the code file queue.c). This package provides the queue manipulation operations necessary to manage a task ready list and implement a variety of priority functions efficiently. This queue management package is also used (with very slight modifications that will eventually be propagated back to the kernel) to provide major support for some of the other tools in the emulator, as we will mention in a later section. It qualifies as one of the "powerful underlying mechanisms" mentioned above.

The third important facility provided by the VHS kernel is data gathering. This is based on the abstraction of an *event record*, a record of a significant event that occurred during exection. Event records are recorded using a macro invocation in the code in places where events of interest occur. For example, just before the call to the task switch routine, the LGOEVENT macro is invoked to record the event—a task switch. The event records are stored sequentially in a buffer in memory while the emulation run. When the experiment is over, the event records stored in the event buffer constitute a record of the significant events that occurred during the experiment. They are then examined to determine exactly what happened. Each event record consists of an event identifier (e.g., Task Switch), a time stamp (32 bits of clock with granularity of approximately 3.5 microseconds), and a parameter (for example, the TID of the task being switched to).

Calls to LOGEVENT can be inserted at any point in the code at which an interesting event could occur. This allows considerable flexibilty—for example, if an experiment raised a question about what is happening in a certain piece of code, a LOGEVENT call could be inserted and the experiment re-run to determine what is happening there. The event record also provides highly precise timings of the intervals between events, and so is a valuable performance measurement aid. It is also useful in debugging, since it provides a trace of the events that occurred proximate to the manifestation of a bug.

Although event logging is not a new idea, this particular facility is the result of the author's experience with similar facilities used in measuring various aspects of the performance of the Unix operating system over a period of four years. This facility is also used in the Alpha kernel, and has already proved valuable for debugging and performance measurement in that context. The tools developed for interpreting the event logs are quite sophisticated, and add to the value of the facility. They are described in section 4.2.2.

4.2.1.3 Sources of Variation

These are simply the factors in emulation that tend to make the results of an emulation run differ from those of a simulation for the same inputs. The sources of variation we have discovered so far fall into three rough categories: timing phenomena, uninteresting overheads, and interesting overheads. We will briefly discuss the members of each category in order.

Timing phenomena include the following effects:

- Clock drift between the VHS and IntGen host processors: This is due to the slight difference in clock rates between different CPUs. It amounts to approximately 0.02% (or about 1 millisecond difference after five seconds). In other word, it is fairly small, relative to task execution times measured in tens of milliseconds and experiment run times on the order of 10 seconds.

- Variations in IntGen: These are due to variability in the time required to schedule the next interrupt and cause the current interrupt to occur. These seem to vary on the order of 0.5 milliseconds for each interrupt. This is still a relatively small effect, but we are investigating means of limiting the variation.

- Variation in "time-wasting" routines: These routines use up CPU time for the user tasks. Variations here are extremely small over long periods.

Uninteresting overheads include the following:

- Unavoidable overheads: The Sun 1.5 CPU requires certain periodic interrupts to coordinate essential housekeeping actions, such as software dynamic memory refresh and bus timeouts. These effects seem to be relatively small, but we are working on characterizing them more exactly.

- Simultaneity of interrupts: In our emulation, we use one processor to emulate possibly more than one I/O device. When interrupts are scheduled (as specified in the interrupt log) to occur simultaneously, they must in fact be dealt with serially by IntGen. This increases interrupt latency slightly in these cases. However, in the real world it is doubtful that many interrupts occur precisely simultaneously.

- Event logging overheads: The act of observation affects the observations, since logging events takes time, which shows up in the experiment results. We have been considering this part of the "normal" system overhead. However, we plan to tune the event logging routine to make it faster (it currently takes around 150 microseconds to log an event). Event logging is not quite as simple as it may sound, since mutual exclusion on the buffer pool indices is required to prevent "lost" events, or skipped event slots in the log.

Interesting overheads involve effects that are common in real systems and are of legitimate interest in understanding how scheduling system actually behave. They include:

- Interrupt handling: It takes a certain amount of time to handle an input interrupt. In [illegible] for very large [illegible] amount of time, but we have the ability to [illegible] for interrupt handling, and this may become an area of [illegible]

- Task scheduling overhead: The very operation of scheduling takes a certain amount of time, which can vary with the scheduling algorithm and possibly with the number of tasks being scheduled, and perhaps with other factors as well. This is one of the criteria that determine how well a scheduling algorithm performs, independent of its theoretical performance.

- Interrupt latency due to running at high IPL: This is caused by operating system code running at high interrupt priority level and locking out I/O interrupts during this time. If an interrupt occurs which the processor is running at high IPL, it is not acknowledged until the processor returns to a lower IPL. Raising the processor IPL is commonly used in operating systems (including VHS) as a means of providing mutual exclusion of access to data structures, and so it is an important effect to quantify. It can also be caused by interrupt service routines themselves running at high IPL. If interrupt service times at high IPL are long, this could become more of a problem, and is an effect worth noting.

- Priority inversion: This effect is observed when interrupts are being serviced at high priority on behalf of a task with low scheduling priority. When this occurs, the scheduling priority discipline has been violated and can result in lower performance of the scheduling algorithm. Instances of this effect have been observed in some experiments already. It is definitely an interesting effect, and raises the possiblity that the way interrupts are handled in typical processors will have to be modified if effective real-time systems are to be built.

4.2.2 Other Tools

This section discusses some of the other tools used in the emulator system. They comprise a substantial amount of code and consumed nearly as much development time as VHS itself.

4.2.2.1 IntGen

This is the program that causes the interrupts listed in the interrupt log to occur. It is structurally simple, a task that causes the current input interrupt to occur, then sets up the Sun 1.5 timer to provide a timer interrupt when the next input interrupt should occur, and then waits. The tricky part of it was to tune the interval timing to take into account the overheads of causing the interrupt and setting up the next interval. IntGen shares some support code with VHS.

4.2.2.2 TSim

This is a special-purpose event-driven simulator used to simulate the input and output DMA scheduling phases during pre- and post-processing phases, respectively. This is a significant piece of code, and is very modular, permitting easy addition of scheduling policies. It also uses the queue management package both for the event list and for the scheduling facilities.

4.2.2.3 Process

This program processes and interprets the event log generated by VHS during an experiment. It is a highly general tool, and its design is based on considerable experience with event logging systems (see Section 4.2.1.2). Its organizational paradigm is a collection of dynamically allocated instances of finite state machine recognizers, which can keep track of individual operation sequences occurring concurrently within the kernel. Although the original design was done with finite state machines in mind, the system as implemented allows more powerful constructs.

This tool provides a highly flexible method of performing very sophisticated analyses of the event log data, which we anticipate will become a valuable capability as the emulator becomes more complex. We also expect it to be valuable in the context of the Alpha kernel. This program also uses the queue package to schedule the activations of the concurrent state machine instances.

4.2.2.4 Miscellaneous Tools

Other tools were also developed:

- *expdef* and *expconf* interpret the experiment specification file to produce data, command, and code files for configuration and operation of the emulator.
- *unbatch* breaks a file of multiple experiment specifications into separate individual experiment specification files and creates directories and command files for later stages of emulator operation.
- *autovhs* runs the emulator by interacting with the VHS and IntGen processors via a serial line interface to control the execution phase of the emulation. With *unbatch*, this allows the complete automation of the emulation for an entire set of test runs. This total automation is accomplished through coordinating the activities of three separate computer systems.
- *time* is a set of several programs derived from the VHS kernel that are used to measure clock drift between processors, calibrate time-wasting routines, and investigate other timing issues of interest.

4.3 Operation and Examples

In this section we illustrate the operation of the emulator by going through one example test run in detail. This will provide a framework for understanding the more general exposition of the previous sections. This example includes the pre-processing, execution, and post-processing phases as well as the initial inputs and final outputs of the run. A diagram of the information flow during emulation is given in Figure 4-3.

We will proceed by working through an actual [illegible] of commands produced by a single emulator test run [illegible] Before the command file and produces a log is

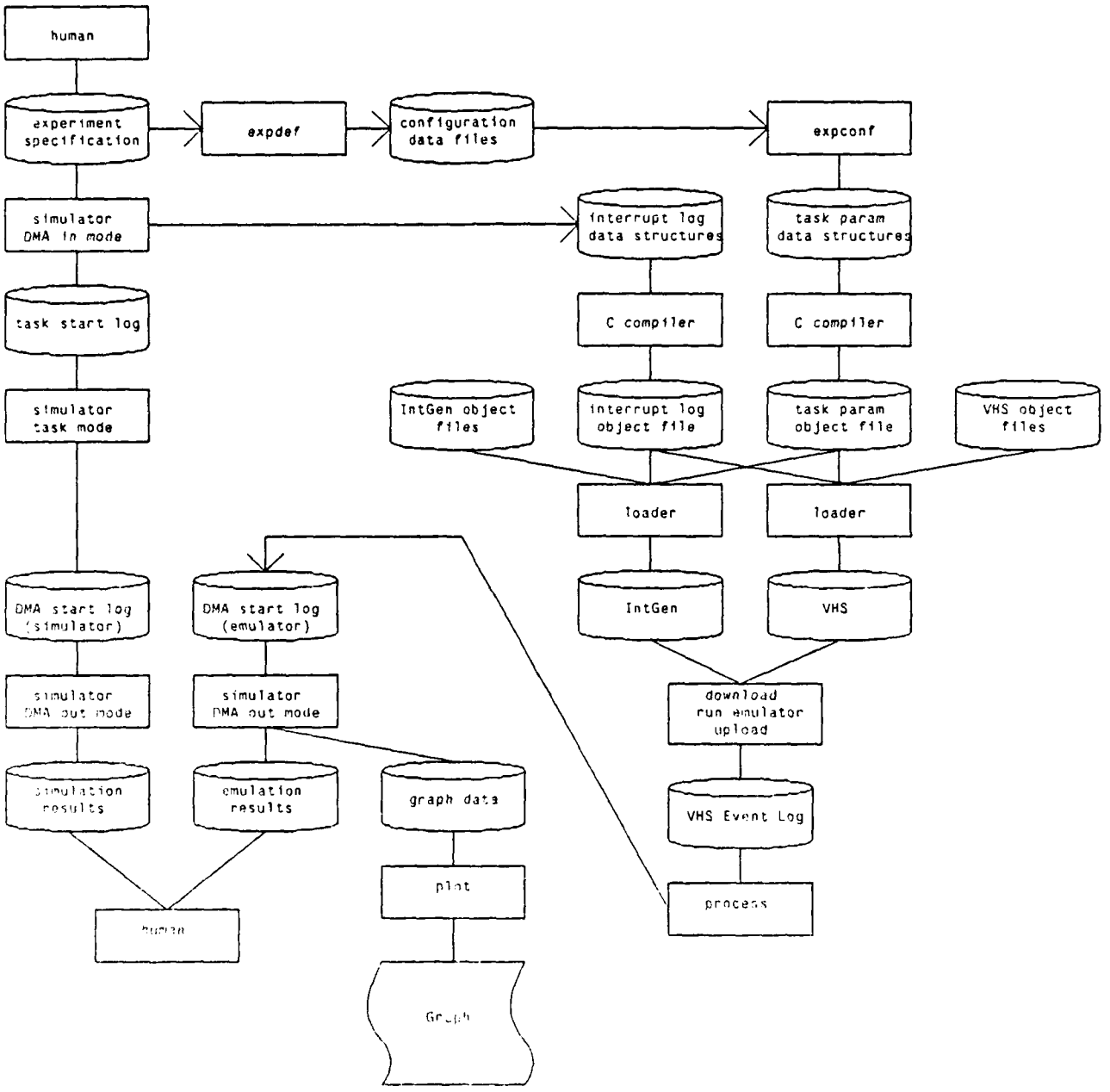


Figure 4-3: Emulator Construction Flow

run, a directory with a file specifying the test must be created. This is usually done by the *unbatch* program, which takes a file consisting of multiple experiment specifications produced by the simulator, breaks it up into individual experiment specification files, and creates a subdirectory for each test.

The subdirectory for a test contains the experiment specification file (called exp.spec) for that test. *unbatch* also produces a command file that performs all the tests by running a program for each test that reads the exp.spec file and makes a command file that performs that particular test, and then executing that command file.

In this case, only one experiment was to be performed, so we do these initial operations manually. Initially, the subdirectory contains only the exp.spec file:

```
policy:              1            1            1
task:   1 :  70 :   0.0 :   10.0 :  18.0 :   6.0 : 0.0 : 0.0
task:   2 :  36 :   0.0 :   10.0 :  18.0 :   6.0 : 0.0 : 0.0
task:   3 : 180 :   0.0 :    4.0 :   6.0 :  12.0 : 0.0 : 0.0
task:   4 : 420 :   0.0 :   10.0 :  12.0 :   6.0 : 0.0 : 0.0
task:   5 : 420 :   0.0 :   16.0 :   6.0 :  12.0 : 0.0 : 0.0
miss: 0
```

The *policy* line gives the scheduling policies for input DMA, task execution, and output DMA, respectively. Policy 1 is rate monotonic, so in this example, all three stages are scheduled by rate monotonic. The *task* lines give the characteristics of each task: the task ID number, the period (all times are in milliseconds), the phase, CPU time, input DMA time, output DMA time, input interrupt service time, and output interrupt service time (currently unused). The *miss* line indicates whether this test missed any deadlines when simulated.

Then we execute the following commands manually (note "t3pap>" is the Unix prompt—in this case, the last component of the current directory path name):

```
t3pap> expdef
computation factor = 1.000000
DMA factor = 1.000000
interrupt service factor = 1.000000
t3pap> chmod +x phase1
t3pap> phase1
```

expdef is a program that reads the experiment specification file and produces a version formatted for other input by other programs:

```
1 70000 0 10000 18000 6000 0 0
2 36000 0 10000 18000 6000 0 0
3 180000 0 4000 6000 12000 0 0
4 420000 0 10000 12000 6000 0 0
5 420000 0 16000 6000 12000 0 0
```

(Most of the programs from this point on deal with times expressed in microseconds.) *expdef* also provides an ability to multiply the task processing times by constant factors; the default value for each of these factors is 1. *expdef* also produces command file (*phase1*) is produced that runs a generic emulation command file with the correct parameters (the scheduling algorithms). We make that file executable, and execute it. The results follow:

```
+ p3 1 1 1
+ expconf -d ../../../temp
```

p3 is the generic emulation command file. (The initial plus signs indicate commands executed by a comand file.) *p3*'s first action is to run *expconf*, a program that produces a configuration file for the experiment, and places them in the specified directory (where it can be conveniently accessed when building the VHS and IntGen systems). This file is essentially a set of C language data declarations equivalent to the experiment specification information, but in a different format.

Now the input DMA phase begins:

```
+ tsim -t 0 -p 1 -g 3
start log at 0, input dma simulation, rate monotonic priority
The hyper-period is 1260000 us.
0 missed deadlines.
```

tsim is the simulator used to simulate input and output DMA scheduling. In this case, it is run with options that specify input DMA scheduling, rate monotonic algorithm, and to produce 3 hyperperiods of simulation data. *tsim* responds by printing out relevant information. The result of the simulation is a file that specifies the completion times of the input DMA operations (which are the initiation times of the tasks) that will be input to the emulator. A sample of its format follows:

```
 18000   18000    2   18000
 36000   18000    1   34000
 54000   18000    2   18000
 60000    6000    3  120000
 90000   30000    2   18000
106000   16000    1   34000
108000    2000    4  312000
...
```

This gives the absolute time of task initiation, the time relative to the last initiation, the task to start, and the time left before the deadline for that task initiation. Another file containing roughly the same information in a different format is produced to provide input to the next command.

tsim is capable of simulating task scheduling as well as input or output DMA scheduling, so it is used to do that simulation as well:

```
+ tsim -t 1 -p 1
start log at 0, task simulation, rate monotonic priority
The hyper-period is 1260000 us.
0 missed deadlines.
+ tsim -t 2 -p 1
start log at 0, output dma simulation, rate monotonic priority
The hyper-period is 1260000 us.
0 missed deadlines.
```

The first execution of *tsim* simulates task execution, and produces an output file of task completion times that provides input to the next execution, which is an output DMA scheduling simulation. These three executions of *tsim* provide, in aggregation, the equivalent of the simulator run on this test, which provides a cross-check of the simulator results, and data for comparison with the emulator results.

The command file proceeds:

```
+ egrep miss exp.spec
miss: 0
+ mv dmaout.sim dmaout.tsim
+ mv result.sim result.tsim
+ awk -f /usr/ses/vhs/cmd/intrlog.awk Gintrlog.def >
    /usr/ses/vhs/temp/Gintrlog.c
```

egrep is a Unix operating system utility that searches for specified text strings in files. The purpose of this command is to print out for a human reader of this log the simulator's prediction about whether deadlines will be missed. Then the intermediate results files for the pure simulation results are renamed so they won't be overwritten by later executions of *tsim* based on data from the emulation.

awk is a Unix operating system utility that provides an interpretive programming language based on pattern matching. This particular *awk* language program converts the interrupt log produced by *tsim* into a file of C language declarations, placed in the same directory with the other generated C language declarations.

```
+ pwd
thisdir=/usr/ses/vhs/exp/even/t3pap
+ cd /usr/ses/vhs/temp
+ cc -c Gintrlog.c
+ cc -c Gexpconf.c
```

These commands remember the name of the current directory and go to the directory with the generated C language configuration declarations, and compile those declarations to object code (cc is the C compiler).

```
+ cd ../intgen
+ make E=/usr/ses/vhs/exp/even/t3pap N=1 intgen
/bin/ld -N -T 4000 -e entry -o intgen ../intgen/coldstart.o
  ../intgen/init.o ../intgen/iointr.o ../intgen/time.o
../intgen/trap.o ../intgen/traprtl.o ../temp/Gintrlog.o
  /usr/lib/libc.a -lc
cp intgen /usr/ses/vhs/exp/even/t3pap/intgen
```

Then the command file changes to the directory with the interrupt generator program and executes a program-building utility (*make*) that causes IntGen to be re-linked with the object files just produced, to configure it for this test. The newly linked IntGen program is then copied to the test directory for later use. Then, the same is done for VHS:

```
+ cd ../sys
+ make E=/usr/ses/vhs/exp/even/t3pap S=1 vhs
/bin/ld -N -T 4000 -e entry -o vhs ../sys/coldstart.o
../sys/event.o ../sys/init.o ../sys/iointr.o ../sys/queue.o
../sys/schpol1.o ../sys/schpol2.o ../sys/sem.o ../sys/spp.o
../sys/syscall.o ../sys/task.o ../sys/time.o ../sys/trap.o
../sys/trapint.o ../sys/user.o ../temp/Gexpconf.o
../temp/Gintrlog.o ../lib/libvhs.1 -lc
/bin/rm -f core
echo "schPol?W1" > ../temp/schpol.adb
adb -w vhs < ../cmd/adbvhs |
    tee /usr/ses/vhs/exp/even/t3pap/vhsadb
                compPct at hex address 74d8
                dmaPct  at hex address 74dc
                intrPct at hex address 74e0
_schPol:        1               =       1
cp vhs /usr/ses/vhs/exp/even/t3pap/vhs
```

The difference with VHS is that a script for the debugger (*adb*) is produced to patch the value of the scheduling policy for this test into the re-linked VHS program. Now, we start the process of running the emulator:

```
+ cd /usr/ses/vhs/exp/even/t3pap
+ rsh archons5 rm -f /pub/temp/ses/vhs.tftp
    /pub/temp/ses/intgen.tftp
+ seg68 -o vhs.tftp -t -d -b vhs
+ rcp vhs.tftp archons5:/pub/temp/ses/vhs.tftp
+ seg68 -o intgen.tftp -t -d -b intgen
+ rcp intgen.tftp archons5:/pub/temp/ses/intgen.tftp
+ rm intgen
```

Back in the test directory, both IntGen and VHS are converted into formats suitable for downloading to standalone processors by the *seg68* program Then they are copied across the network to a machine that has an Ethernet connection to the host processors, preparatory to downloading and execution. At this point, *autovhs* is run. This program communicates with the host processors via a serial line interface to execute the commands to the processors PROM monitors to cause VHS and IntGen to be downloaded, executed, and to upload the VHS core image[1]. The following portion of the log has been edited slightly for readability:

---

[1] [illegible]

```
+ autovhs
**resetting vhs***
>k 3
Self Test completed successfully.
Sun Workstation Monitor - 0x040000 bytes of memory
Processor: 68010; No Parallel Keyboard;
    PROM 1&3: Rev AC; PROM 2&4: Rev AC
>**resetting intgen***
>k 3
Self Test completed successfully.
Sun Workstation Monitor - 0x040000 bytes of memory
Processor: 68010; No Parallel Keyboard;
    PROM 1&3: Rev AC; PROM 2&4: Rev AC
>***download vhs***
>j
cmd: r
start: 0x4000
size:
filename: /pub/temp/ses/vhs.tftp
23600 Bytes Received from '/pub/temp/ses/vhs.tftp'
>***download intgen***
>j
cmd:r
start: 0x4000
size:
filename: /pub/temp/ses/intgen.tftp
26780 Bytes Received from '/pub/temp/ses/intgen.tftp'
>***start vhs***
g 4000
VHS version 0.0 on Sun-1
262144 bytes of usable memory:
etext 29408 edata 34780 end 39984
task 1: comp = 10000 dma in = 18000 dma out = 6000 intr = 0
...<lost characters due to lack of
hardware flow control on the lines>...
Ready ***start intgen***
g 4000
intgen version 1.0 on Sun-1
Init done
Goodbye, cruel world. <from IntGen>
Goodbye, cruel world. <from VHS>
Done ***upload core image***
>j
cmd: s
start:
size:
filename: /tmp/vhs.core
262144 Bytes Sent to '/tmp/vhs.core'
***done***
```

Then the post-processing begins:

```
+ rcp archons5:/tmp/vhs.core .
+ slurp
+ process
##############
processed 749 events
```

The VHS core image uploaded to the archons5 machine is copied back to this machine and a program (*slurp*) extracts the event log from the core image. Then *process* processes the events and produces a list of task completion times (output DMA initiation times) for input to the output DMA simulation. It also produces some other outputs, such as files of points for plotting time-lines (which will be discussed later), and some debugging output, such as a human-readable log of the events. The following is an example of the format of that log:

```
<ev #><ev ID> <ev desc><ev parameter><abs event time> <rel.>
24: 11: I/O intr arrives      : 2: t: 4489346 r:     18854
25: 12: Task switch           : 2: t: 4490231 r:       885
26: 10: computation complete  : 2: t: 4500439 r:     10208
27: 12: Task switch           : 0: t: 4501064 r:       624
28: 11: I/O intr arrives      : 1: t: 4505335 r:      4270
29: 12: Task switch           : 1: t: 4506116 r:       781
30: 11: I/O intr arrives      : 4: t: 4507314 r:      1197
31: 10: computation complete  : 1: t: 4517158 r:      9843
```

The post-processing proceeds:

```
+ tsim -t 2 -p 1
start log at 0, output dma simulation, rate monotonic priority
The hyper-period is 1260000 us.
2 504000 540455 missed deadline by 455
2 576000 612382 missed deadline by 382
2 1764000 1800135 missed deadline by 135
2 1836000 1872113 missed deadline by 113
2 3024000 3060074 missed deadline by 74
2 3096000 3132261 missed deadline by 261
6 missed deadlines.
+ mv dmaout.sim dmaout.em
+ mv result.sim result.em
+ rm vhs.core vhs.tftp intgen.tftp vhs
```

The output DMA simulation is run, with input from the emulation, and it shows that some deadlines were missed, but not by much—around half a millisecond at most. Finally, some cleanup of files no longer needed is performed.

Both *tsim* and *process* produce files of point values that are specifically designed to be useful in plotting a time line of the task and DMA executions. The format of such a point file is straightforward:

```
18 , 2
36 , 1
54 , 2
60 , 3
90 , 2
...
```

Another program, *gcmd*, takes these files of points and produces a command file for a plotting program called *plot*[2] that will produce a time line. One can specify the interval of time (in milliseconds) to be printed, and how many milliseconds should be plotted for each page of the graph. An example of the graphical output is shown in Figure 4-4. (This is the first page of the output produced by *gcmd* by default for the above run. Note that time is on the X axis and task ID is on the Y axis. For each task, input DMA times are plotted slightly above task execution times, which are slightly above output DMA times. The symbol "[" refers to the intitation of a task, and "]" marks the completion of a task (either input DMA, CPU, or output DMA). By convention, the idle task has task ID 0.

[2] Developed at Carnegie-Mellon University by Ivor Durham.

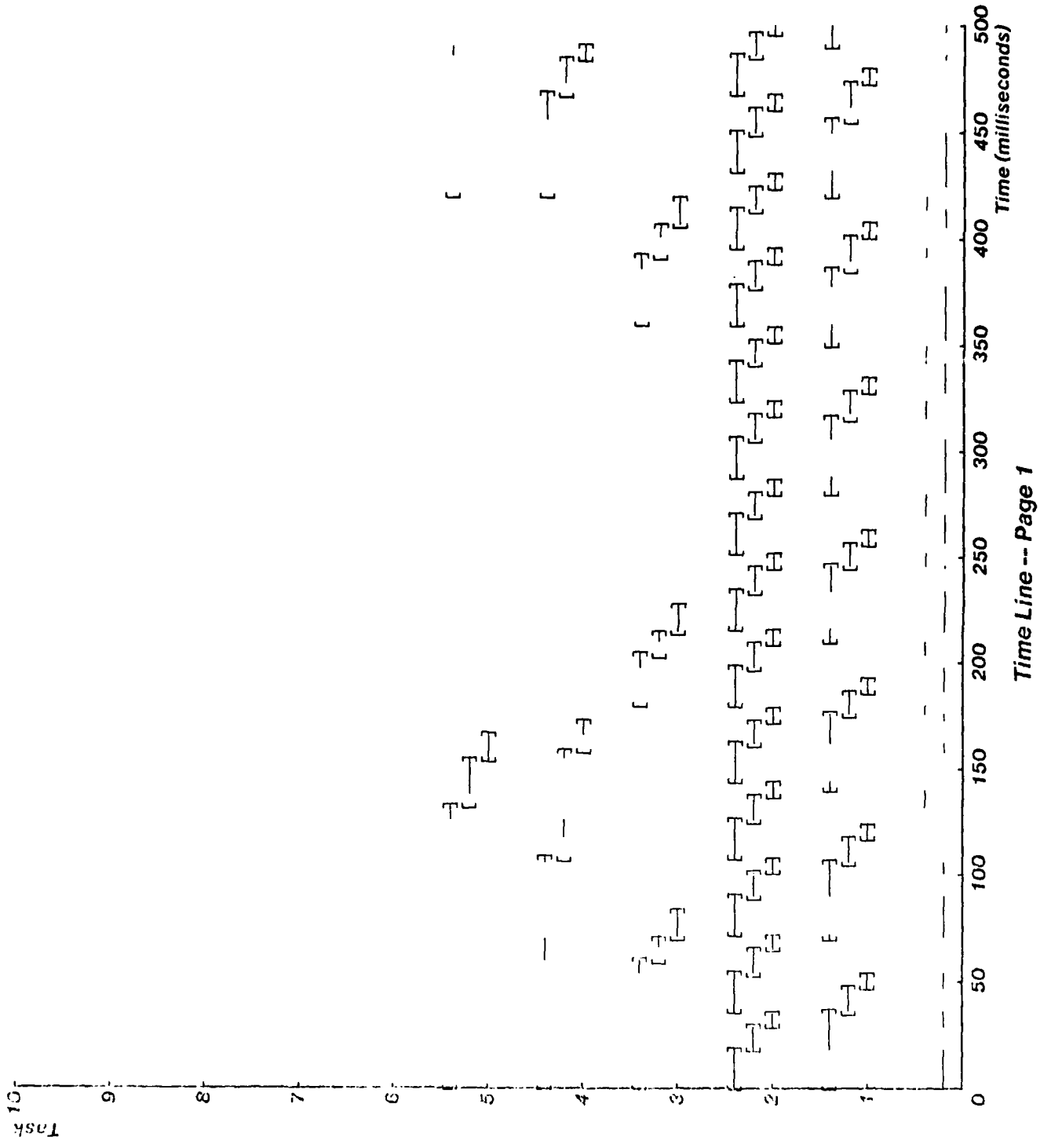


Figure 4-4: Graphical Output Example

5. Summary of Pilot Study

This year was largely devoted to conceptualizing the I/O scheduling problem and to building tools to study it. Nevertheless, a pilot study was undertaken to learn about I/O scheduling. The results of this pilot study were quite clear cut and promising. We summarize the pilot study, the results and the implications that can be drawn.

5.1 Pilot Study

Traditionally, the theory of processor scheduling is based upon a worst case analysis approach. Such an approach is particularly useful when there exists a scalar measure of performance, and when the worst case performance bound is sufficiently high from an application point of view. If a worst case bound is deemed to correspond to poor performance, one may still use the algorithm if the worst case seems unlikely to arise in practice. A statistical approach is generally more appropriate when the measure of performance involves many factors or when the average performance is sufficiently different from the worst case performance. In the case of scheduling periodic tasks with I/O, the performance of scheduling algorithms is measured by the ability to schedule a task set. The task set is often measured by three parameters: the input DMA load, the output DMA load and the CPU load. It is rather easy to create worst case situations corresponding to low utilizations for which deadlines will not be met. These are especially unrepresentative task sets, thus a statistical approach is far more appropriate for this investigation. Having opted for a statistical approach, we still must determine a method a creating task sets to test the various algorithms. The approach taken was to create task sets which had some generality.

5.2 Experimental Design

A general goal of the pilot study is to create task sets, select certain scheduling algorithms and determine if the algorithms can meet all the timing requirements of the task set. By doing this with a large number of task sets (using either the emulator or the simulator) one obtains a picture of the performance of the scheduling algorithms being studied. Clearly, there is an infinite number of possible task sets that might be faced. Once a selected subset can be actually studied. We want to pick as large a subset as possible, and one which will allow us to make useful conclusions. There are several technical issues which arise. First, since we had a fixed amount time and money to perform simulations, we attempted to devise a method which would allow us to maximize the number of cases that could be examined. Second, we had to choose cases which were in some sense representative. Given that the halting theorem[3] requires us to carry out a run over as many as [illegible]

[illegible] page 2

should try to keep the hyperperiods as short as possible. If one picks task periods at random, then extremely long hyperperiods are a likely consequence. for example, if one picked the four periods 247, 258, 123, and 643, then the hyperperiod would be 1,185,527,382. If one considered a task set with ten tasks, it is likely that the resulting hyperperiods would be too long to be run at all. It is, therefore, important to design a method which will create a relatively short hyperperiod. We used the following approach. We began with a set of 7 prime numbers, {2,2,2,3,3,5,7}. All task periods were drawn by randomly selecting a subset of these elements and multiplying them together to create a period. For example, if 2, 3, and 7 were picked, then that task would have period 42. Using this method, we insure that the hyperperiod is always less than or equal to 2520. We always used ten tasks in the task set. There is an important concern about this approach. We wonder if it can generate a "bad" task set. A bad task set is one with low utilization that cannot be scheduled. The answer to this is "yes" in the case of pure processor scheduling. In that restricted case, bad task sets are those in which the relative periods are in the ratio of 1 to 1.40 [Liu 73]. One can easily generate such ratios using these prime numbers. It is still necessary to have the relative computation times be chosen in a specific way to obtain a bad set; however, we utilize a uniform random number generator to ensure that these relative computation times can be obtained. Therefore, this method has the potential to generate both bad and good task sets in the pure processor scheduling case. In the case of I/O scheduling, it is reasonable to draw the tentative conclusion that our method will also be able to generate bad task sets in this broader situation. This follows because it can generate a wide range of period ratios and uses a uniform random number generator to determine DMA and computation utilization.

Given that we wanted to run as many task sets as possible, the simulation program was constructed to take an initial task set and systematically scale up the utilizations at each of the three phases until a task set was created which could no longer be scheduled. This allowed us to determine a "breakdown point" for each algorithm for each basic task set. This set of breakdown points is especially valuable, because it offers a direct comparison of the performance of the competing algorithms under fixed conditions. It was these breakdown points which offered a rather clear and consistent picture of the behavior of the algorithms.

Given we were looking for breakdown points for a set of algorithms over randomly generated task sets, it became useful to further refine our view of task sets. Clearly, the importance of I/O scheduling depends in part on the amount of I/O associated with the tasks. If tasks are "compute bound" and have relatively small amounts of I/O, then I/O scheduling will be relatively unimportant. Similarly, if tasks are "I/O bound", then I/O scheduling will be of much greater importance. It is, therefore, useful to introduce the notion of "I/O bound" and "CPU bound" into the task set generation process. We accomplish this by specifying a total of relative utilizations for each of the three phases, for

example, (1,5,1). This means that we are creating tasks in a task set whose CPU utilizations tend to be 5 times as large as either their input DMA or output DMA requirements. It should be noted that tasks are created at random, and it is possible for one or two of the tasks so created to actually require more input and output time than CPU time; however, on average the tasks will require 5 times as much CPU time as either input or output time.

The pilot study consisted of two phases. First, we made runs for 5 cases: (1,5,1), (3,5,3), (1,1,1), (7,5,7) and (9,5,9). These span the spectrum of "compute bound" to "I/O bound". The pilot study consisted of running each case assuming no deadline postponement and then again assuming one deadline postponement. Second, we made a series of runs for the (7,5,7) case to develop a notion of the average case behavior of the scheduling algorithms used. Finally, all of this was done for 5 scheduling algorithms: FFF, FRF, RRR, DDD, and PPP where F denotes FIFO, R denotes rate monotonic, D denotes dynamic deadline and P denotes propagated deadline. The first symbol refers to input DMA scheduling, the second to CPU scheduling, while the third refers to output DMA scheduling.

5.3 Results of the Pilot Study

5.3.1 The importance of I/O scheduling

The pilot study consisted of generating task sets at random and subjecting them to the I/O and CPU scheduling algorithms mentioned earlier. A typical task set consisted of ten tasks with randomly generated periods and phases. As was discussed in the previous section, the input DMA, output DMA and CPU computation times were chosen in a way to create a task set characteristic of I/O bound tasks, CPU bound tasks and mixtures of the two. A set of attached figures (Figures 5-1 to 5-5) summarize a range of cases. As an example, consider Figure 5-1. In this case, tasks were generated randomly with relative utilizations of one for both input DMA and output DMA and five for the CPU. Consequently, these tasks tend to be compute bound. Once the periods, utilizations and phases were fixed, the utilizations were scaled up until a deadline was missed using a particular scheduling algorithm. The same procedure was carried out using the same scheduling algorithms but with one deadline postponement. In each of the figures, FFF refers to FIFO being used at all three stages of scheduling, while FRF uses FIFO for I/O scheduling and rate monotonic for processor scheduling. RRR, DDD and PPP refer to rate monotonic, dynamic deadline and propagated deadline being used at all three stages of scheduling respectively. In Figure 5-1, we observe that the FIFO scheduling algorithm missed a deadline when the bottleneck utilization (the processor in this case) reached 85%. With one deadline postponement, the bottleneck utilization was raised to 89% before a deadline was missed. The results for FRF and PPP were better since 90% bottleneck utilization was reached

before a deadline was missed. The more sophisticated algorithms (DDD and PPP) were able to meet all deadlines when bottleneck utilization was raised to 100%.

The situation is changed somewhat as the I/O utilization is increased and I/O scheduling becomes more important. Figure 5-2 considers a case with ten randomly generated tasks drawn from a distribution whose relative I/O to CPU utilizations are 3 to 5. The CPU is still the bottleneck, but the FFF and FRF algorithms miss deadlines at 23% and 37% bottleneck utilization. One deadline postponement raises these to 38% and 70% respectively; however, this is far inferior to the 100% bottleneck utilization achieved by RRR, DDD and PPP with no deadline postponement! This situation persists in Figure 5-3. Here, ten tasks are randomly generated from a distribution with equal I/O to CPU utilization (1,1,1). The FFF and FRF miss deadlines at 35% bottleneck utilization and reach 63% and 70% with one deadline postponement. The RRR, DDD and PPP algorithms handle 92%, 98% and 98% with no deadline postponement. Figure 5-4 presents the case of (7,5,7) where I/O dominates the CPU. The results are particularly interesting. Again, the sophisticated algorithms dominate the FFF and FRF. Here, we observe the benefits of the PPP algorithm which handles 93% with no postponement. Finally, Figure 5-5 shows clearly the vital importance of I/O scheduling. Tasks are generally I/O bound. Here FFF and FRF were able to handle only 17% bottleneck utilization (input and output DMA) and only 30% with one deadline postponement. This terrible performance was in marked contrast with RRR, DDD and PPP which handled 98% with no deadline postponement.

The figures are typical of the behavior observed. FIFO scheduling for I/O is highly erratic. It is possible that such an algorithm could achieve a decent breakdown utilization (equivalent to RRR) for one phasing but for another to miss deadlines at a very low utilization. Clearly I/O must be scheduled just as the CPU if its utilization is at all significant. The important observation is that a simple algorithm such as RRR achieves an excellent breakdown utilization.

To sharpen our understanding of the relative behavior of these algorithms, we performed a second simulation experiment. This consisted of generating ten sets of ten tasks using a (7,5,7) relative utilization. The phase was chosen to be zero for all tasks, and no deadline postponement was used. Each of the ten task sets was scheduled by the five algorithms under consideration. We present the results in the following table.

TABLE 1

CASE	FFF	FRF	RRR	DDD	PPP
1	.206	.206	.897	.897	.929
2	.274	.274	.858	.951	.952
3	.318	.342	.927	.973	.973
4	.392	.392	.923	.964	.964
5	.409	.409	.859	.961	.961
6	.415	.415	.861	.940	.952

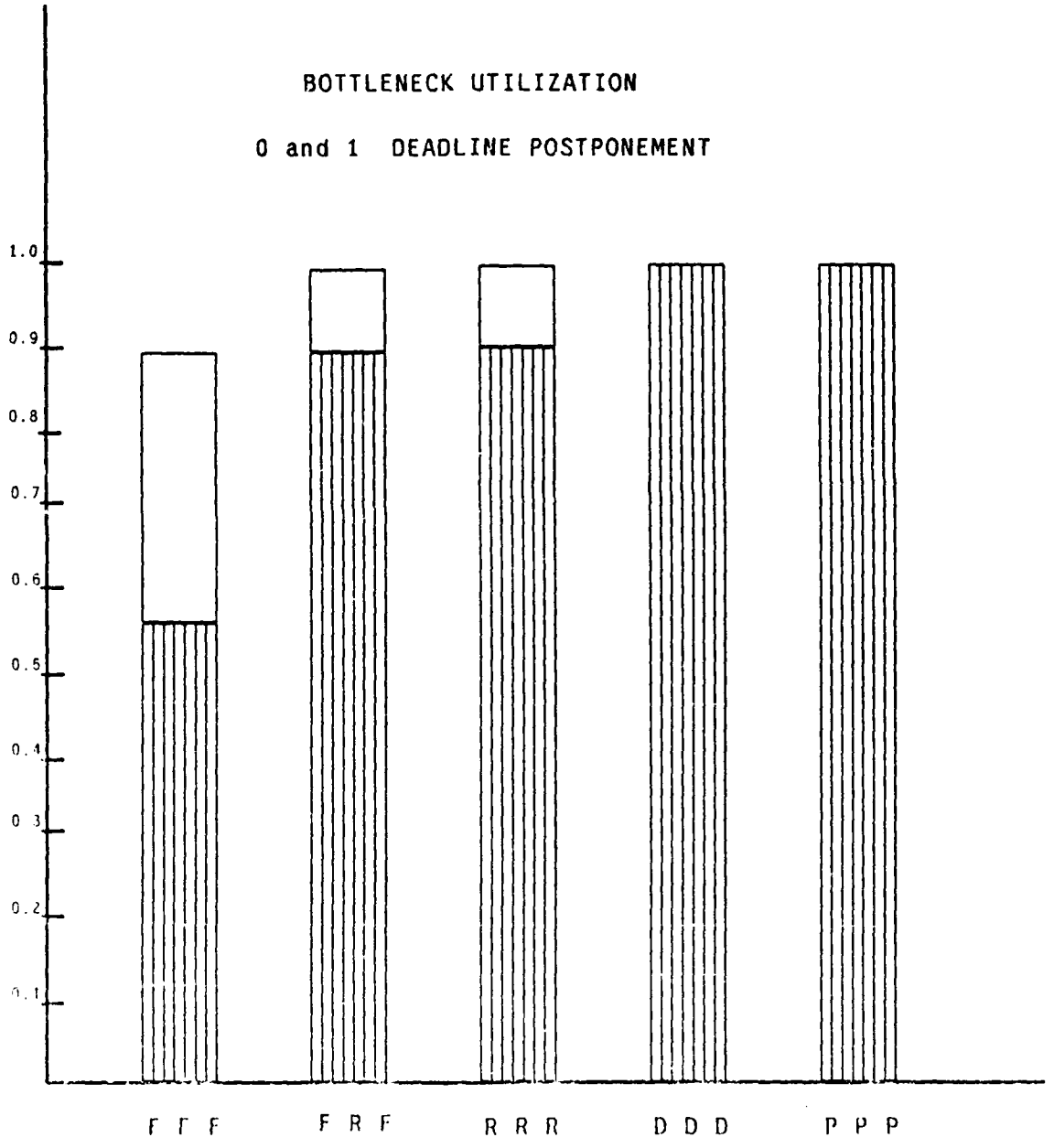


Figure 5-1: 10 Tasks, U = 1, phase = 0

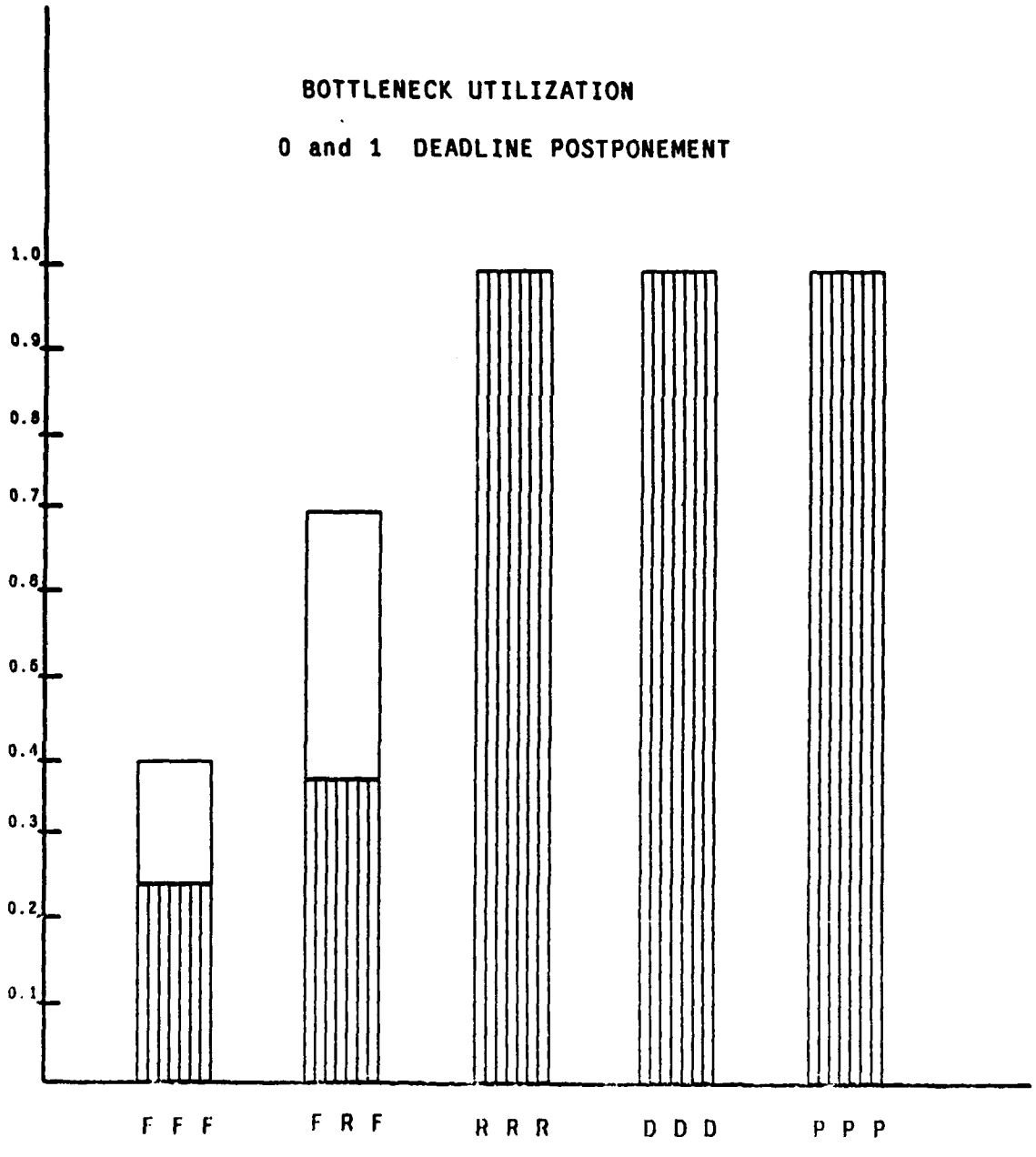


Figure 5-2: 10 Tasks, 3 5 3, phase = 0

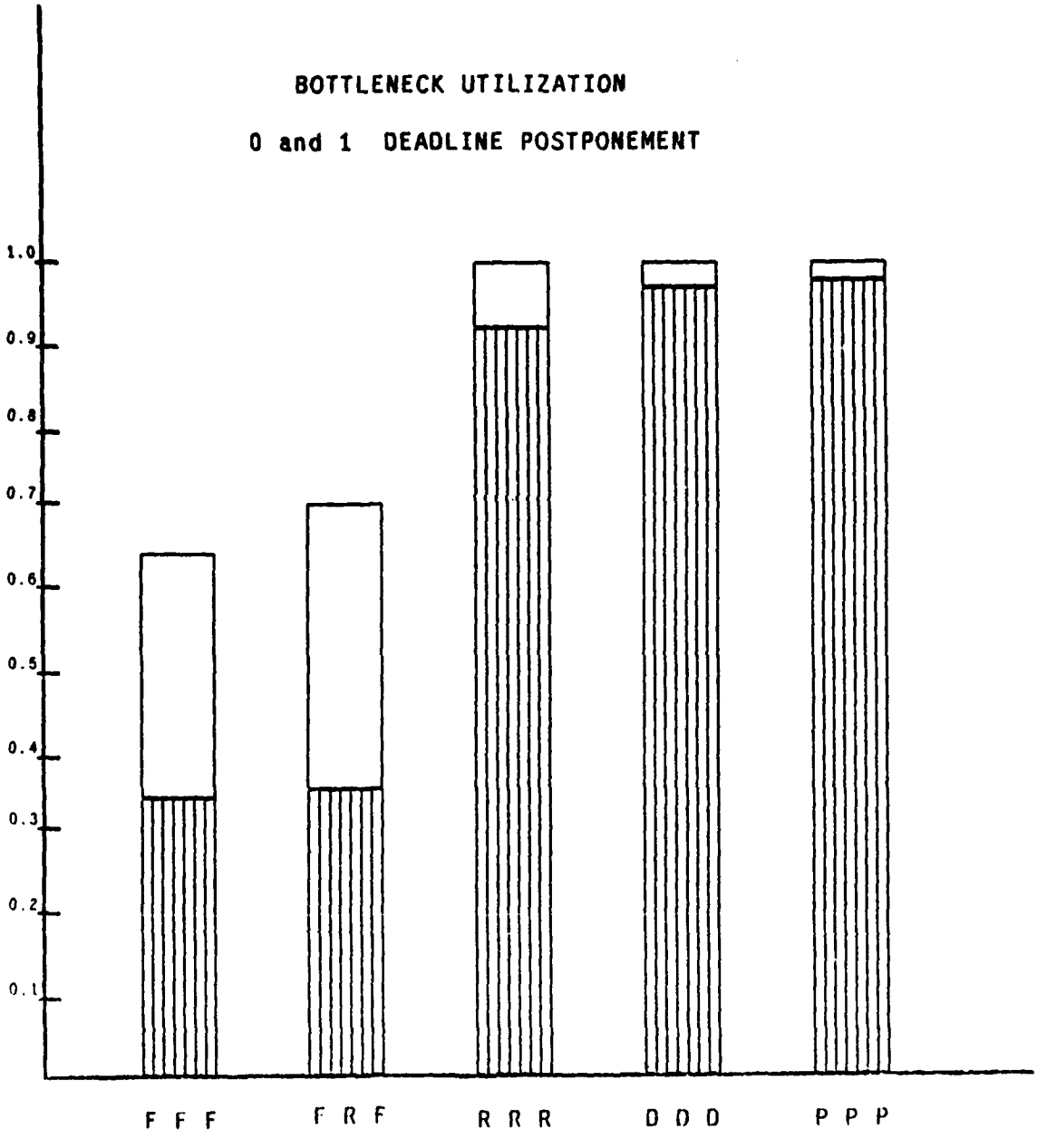


Figure 5-3: 10 Tasks, 1 1 1, phase = 0

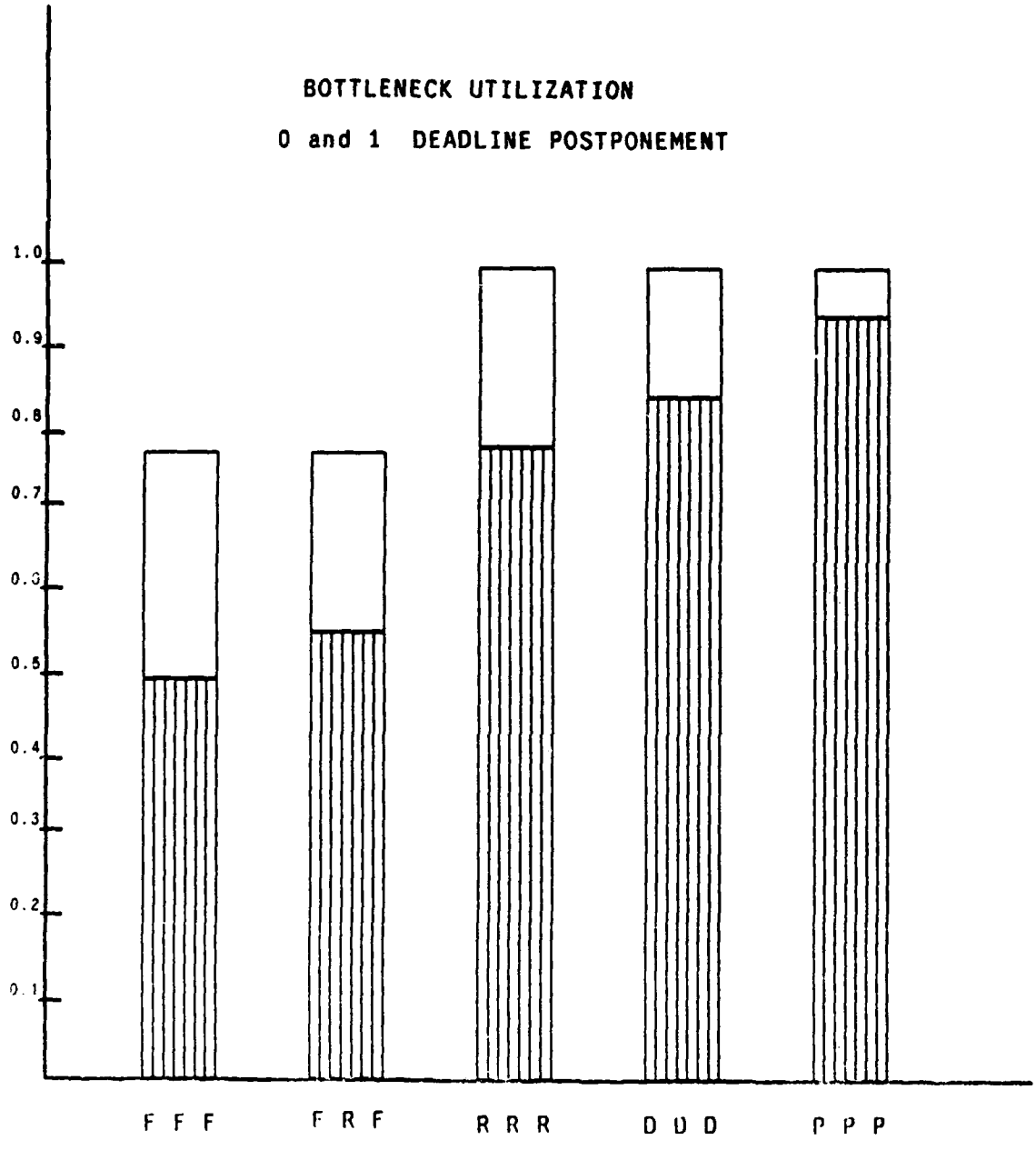


Figure 5-4: 10 Tasks, 7 5 7, phase = 0

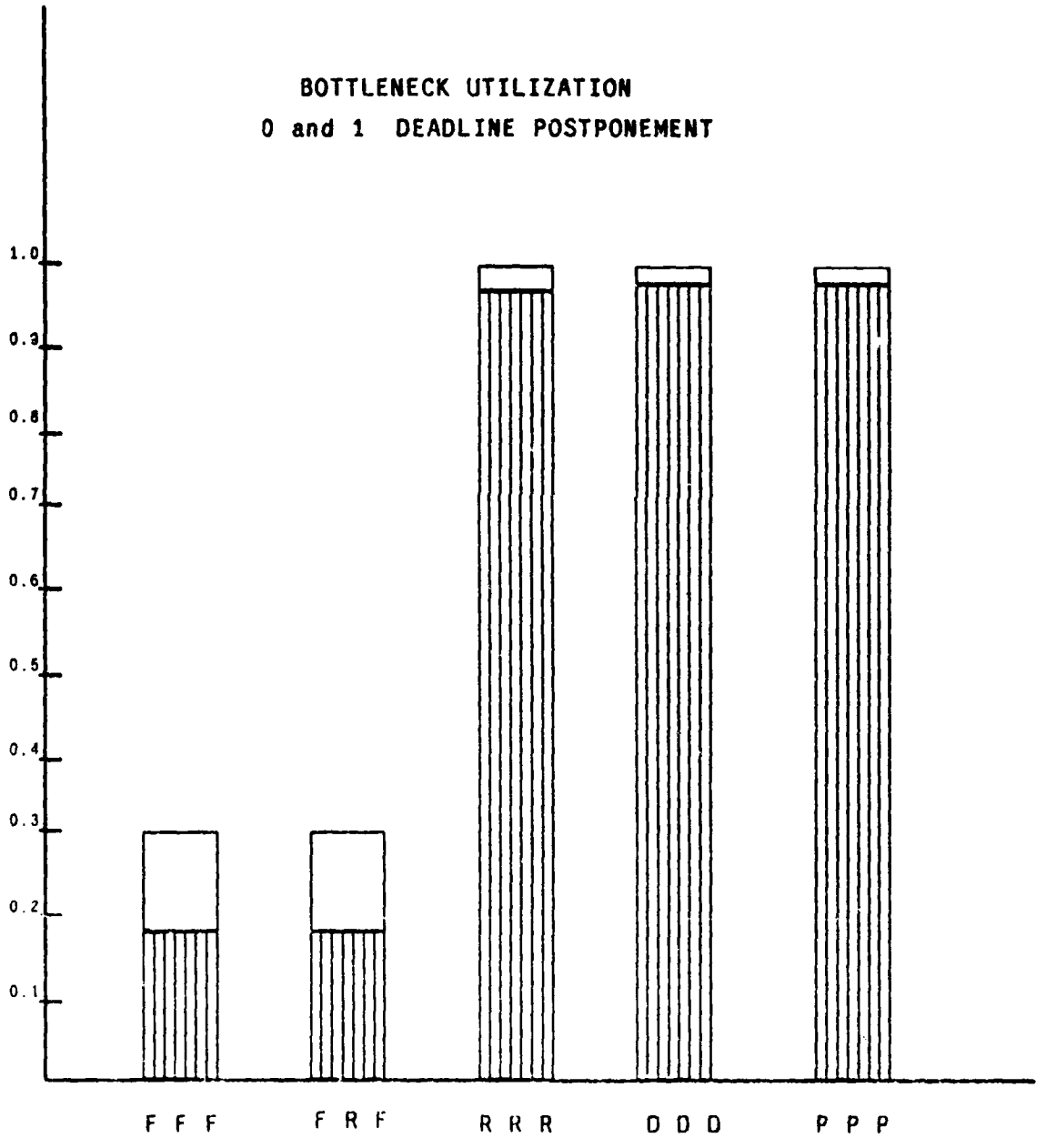


Figure 5-5: 10 Tasks, 9 5 9, phase = 0

7	.425	.425	.934	.978	.978
8	.445	.498	.890	.890	.948
9	.467	.513	.872	.954	.954
10	.554	.554	.926	.983	.987

The table clearly illustrates the drawbacks to using an algorithm such as FIFO and the benefits of using the rate monotonic algorithm. FIFO has an average performance around 41% with a range of 20% to 55%. This is in marked contrast to the behavior of RRR. It has a rather tight range of performance making its behavior fairly predictable. We found its range to be from 85% to 93% with an average of about 89%. Had more than 10 cases been run, it is likely that the range would have been wider; however, from our experience the average would be about the same. The 90% figure is a rough measure of the performance of the RRR algorithm in the (7,5,7) case. The minor differences between the performance of FFF and FRF simply arises from the fact that these task sets tend to be I/O bound. Adding a good scheduling algorithm to the CPU has very limited benefits, since the I/O tends to be the bottleneck. One can observe that FFF and FRF have a wide variance. This is in keeping with our experience. It is possible for FFF and FRF to do adequately (rarely) and miserably (more commonly).

The behavior of the more sophisticated algorithms such as DDD and PPP is even better than RRR. The DDD algorithm has a range of 89% to 98% with an average around 95%. The PPP algorithm is the best with a range of 93% to 99% with its average around 95%. It is clear that these algorithms dominate RRR in terms of pure performance, but the gap is quite small. Given the simplicity of RRR and the relative ease of implementation compared with DDD and PPP, these data offer a compelling case for the use of RRR.

5.3.2 The bottleneck utilization result

The I/O scheduling problem contains three scheduling components. As such, it has not been treated in the literature. Surprisingly, if one computes the utilization at each of the three stages and if one of these three utilizations dominates the other two, then the schedulability of the task set is determined by the bottleneck stage. This means that the schedulability of a task set with a single bottleneck can be determined as if the scheduling were a single stage. This reduces a three dimensional scheduling problem to a single dimension and allows one to use known results in the literature for rate monotonic or deadline scheduling.

5.3.3 Performance of Algorithms

The three stage rate monotonic algorithm (rate monotonic used at each of the three stages) on average ensures that all deadlines will be met when the bottleneck utilization is well above 80%. With one deadline postponement, the deadlines are on average met when this utilization is well above 90%. The results of the pilot study actually show that these are somewhat conservative percentages. It is not uncommon to see RRR with no deadline postponement achieve over 90% utilization and 100% utilization with one deadline postponement.

The deadline and propagated deadline algorithms improve the average threshold at which deadlines are missed to well above 90% and to well above 95% when a single deadline postponement is allowed. Again, these are fairly conservative percentages based on the pilot data. It is typical to see DDD and PPP meet all deadlines at 95% bottleneck utilization and to handle 100% with one postponement.

There is one caution that should be mentioned. The pilot study consisted of data sets which tended to be characterized by small tasks. That is, the largest task utilization divided by the task set utilization was small, say 20%. This situation tends provide a favorable scheduling environment, one in which high utilizations are possible. The sophisticated algorithms tend to offer the most benefit when the task set is unbalanced, that is when several tasks have tight timing requirements, then a sophisticated algorithm may be necessary to handle the scheduling, even at moderate utilizations. When the task set is balanced, slight scheduling errors caused missed deadlines only at relatively high utilizations.

5.4 Conclusions

The finding presented in the previous section must be regarded as tentative. In 1986, we intend to carry out a much more thorough exploration. Furthermore, we intend to investigate other issues including:

- Aperiodic response times,
- Algorithmic behavior under transient overload conditions, and
- The effects which occur when input DMA, CPU, and output DMA conflict in some way, say through "cycle-stealing".

In spite of the tentative nature of the study, the conclusions drawn in the previous section have important implications for hardware and software designs. These include:

- The rate monotonic should be used on the I/O channel processors. Its breakdown utilization is very high, and it is relatively easy to implement. The further gains from a more optimal scheduling algorithm are likely outweighed by the cost of their implementation.

- The existing I/O channel processor architecture found in most computers makes the implementation of I/O scheduling difficult. Such architecture should be modified to at least to support the rate monotonic scheduling algorithm.

- When scheduling I/O, the DMAs should be made as preemptable as possible, for example by using a quanta transfer method.

- Sophisticated algorithms are useful when large tasks must be scheduled in a single task set (large tasks are single tasks with little slack). Simple algorithms are perfectly adequate when there are many small processes to be scheduled even if their total utilization is high. In addition, the emulator clearly shows that when a task set contains some large tasks, then effects such as the priority inversion created by interrupts can have untoward effects on scheduling, i.e. interrupts can cause deadlines to be missed.

- Deadline postponements help in allowing great utilization with no missed deadlines. They do, however, entail additional buffering and can cause increased phase delay. Their greatest value seems to be in smoothing out transient overloads without missing any deadlines. This point will be explored more fully next year.

References

[Lawler 81] Lawler, E. L.
Scheduling Periodically Occuring Tasks on Multiprocessors.
Information Processing Letters , February, 1981.

[Leung 80] Leung, J. Y. and Merril M. L.
A Note on Preemptive Scheduling of Periodic, Real Time Tasks.
Information Processing Letters , Nov. 1980.

[Liu 73] Liu, C. L. and Layland J. W.
Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment.
JACM , 1973.

[Martel 82] Martel, C.
Preemptive Scheduling with Release Times, Deadlines and Due Times.
JACM , July, 1982.

[Mok 83] Mok, A. K.
Fundamental Design Problems of Distributed Systems For The Hard Real Time Environment.
PhD thesis, M.I.T., 1983.

4.8 Modular Concurrency Control and Failure Recovery

Modular Concurrency Control and Failure Recovery

--- Consistency, Correctness and Optimality

Lui Sha

Department of Electrical and Computer Engineering
Carnegie-Mellon University

This work was sponsored in part by: the USAF Rome Air Development Center (RADC) under contract F30602-81-C-0297; the US Naval Ocean Systems Center (NOSC) under contract N66001-81-C-0484; and the IBM Corporation under contract YD-289658. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of RADC, NOSC or IBM.

Acknowledgements

This thesis would not have existed were it not for aid and comfort from many sources. I can hope to mention only a few important names. To all of the others who have helped me, please accept my thanks.

I would like to thank my advisor, Prof. E. Douglas Jensen, for his guidance, encouragement and support. His vision of a decentralized computer system has created the Archons project, which provides us with both a new and exciting area of research and a stimulating and productive working environment. I commend Prof. John P. Lehoczky for his great endurance in reading many poorly written drafts of my early ideas and for his prompt response to each of my attempts to clarify those ideas. His penetrating comments are a source of inspiration to me. Prof. Martin S. Mckendry's insight into the system aspects of concurrency control is always clear and fresh. Although I met him late and have had relatively few discussions with him, every discussion led to new clarification of my ideas. It is a great pleasure to thank Prof. John Shen for encouraging me to do independent research and for providing me with many advices and support.

I would like to thank Dr. Hide Tokuda, Raymond Clark and Douglass Locke for many productive discussions of ideas related to my theory and their object oriented approach of designing the decentralized operating system ArchOS. Such discussions are particularly helpful in providing guidance for future work. I would also like to thank Prof. Richard Rashid and Prof. Alfred Spector for their helpful comments and to thank Cynthia Hibbard for teaching me the skills of writing.

My special thanks goes to my wife, Shirley, who is always most understanding. Without her encouragement and support, I probably would never have made it. I would also like to thank my parents for their unending confidence in my ability to accomplish important goals in life, and for providing a strong incentive to do so.

Abstract

A distributed computer system offers the *potential for a degree of concurrency*, modularity and reliability higher than that which can be achieved in a centralized system. To realize this potential, we must develop provably consistent and correct scheduling rules to control the concurrent execution of transactions. Furthermore, we must develop failure recovery rules that ensure the consistency and correctness of concurrency control in the face of system failures. Finally, these scheduling and recovery rules should support the modular development of system and application software so that a transaction can be written, modified, scheduled and recovered from system failures independently of others.

To realize these objectives, we have developed a formal theory of modular scheduling rules and modular failure recovery rules. This theory is a generalization of the classical works of serializability theory, nested transactions and failure atomicity. In addition, this theory addresses the concepts of consistency, correctness, modularity and optimality in concurrency control and failure recovery. This theory also provides us with provably consistent, correct and optimal *modular concurrency control and failure recovery rules*.

Currently, this theory is being used in the development of the ArchOS decentralized operating system at the Computer Science Department of Carnegie-Mellon University [Jensen 84, Tokuda 85].

1. Introduction

Distributed computer systems offer the potential to develop a decentralized operating system that supports highly concurrent and reliable computations that are needed in many large scale distributed real time control systems. One important objective of the ArchOS decentralized operating system is to realize this potential via its atomic transaction facility at the kernel level [Jensen 84]. To this end, we must develop provably consistent and correct concurrency control and failure recovery mechanisms. Concurrency control mechanisms must ensure the consistency and correctness of concurrent executions of a system. Failure recovery mechanisms must be able to maintain the consistency and correctness of concurrency control in the face of system failures. These two types of mechanisms should also support the modular development of system and application software so that a transaction can be written, modified, scheduled and recovered from system failures independently of other transactions. The objective of this thesis is to provide a theoretical foundation for the development of such modular concurrency control and failure recovery mechanisms.

This chapter is organized as follows. In Section 1.1 we review related works and discuss the main contribution of this thesis. In Section 1.2 we give an overview of the main concepts and results of this thesis, and in Section 1.3 we describe the organization of this thesis.

1.1 Related Works and The Main Contribution of This Thesis

In this section, we review the important developments in concurrency control and failure recovery theories and point out the main contribution of this thesis. In Section 1.1.1, we review related works and in Section 1.1.2 we discuss the main contribution of this thesis.

1.1.1 Related Works

The review of related works is divided into three parts. In Section 1.1.1.1, we state the fundamental assumptions used by concurrency control and failure recovery theories and in Section 1.1.1.2, we discuss some of the important developments of serializability theory and failure atomicity. Finally, in Section 1.1.1.3 we examine some recent development in non-serializable concurrency control and failure recovery theories.

1.1.1.1 Fundamental Assumptions

In concurrency control, two properties of transactions are of fundamental importance: consistency and correctness. By "consistent", we mean that the results of executing a set of transactions satisfy database consistency constraints, and by "correctness" we mean the satisfaction of the transaction post-conditions. In failure recovery, an important concept is "clean and soft" failures [Bernstein 83]. A system failure is said to be clean and soft if its effect can be modelled by a network of computers some of which stop running and lose the content of their main memories. However, the database, including the portion of the database residing in the stable storage used by the failed computers, is not affected by the failures. The stable storage is an idealized model of a network of reliable disks or other reliable storage devices whose probability of failure is so small that such failures can be ignored by the applications in question. In the following discussion, we assume that each transaction in the system is consistent and correct when executing alone. Furthermore, we assume that failures are clean and soft.

1.1.1.2 Serializability Theory and Failure Atomicity

The classical works on concurrency control and failure recovery are known as *serializability theory* and *failure atomicity* respectively [Bernstein 79]. Serializability theory states that a schedule is said to be serializable if the computations performed by any transaction under this schedule are equivalent to those under some serial schedule. The consistency and correctness of serializable schedules follow immediately from the definition of serializable schedules and from the consistency and correctness assumptions about individual transactions. The most commonly used scheduling mechanism for enforcing serializability in concurrency control is the *two phase lock* protocol [Eswaran 76], which states that a transaction cannot acquire any new lock if it has released any of the acquired locks.

The failure recovery rule used in the context of serializable schedules is called failure atomicity. This rule states that under a serializable schedule a transaction either commits all its executed steps at the end of the transaction or aborts these executed steps. Committing a sequence of executed steps means non-divisibly transferring the results of computation performed by this sequence of steps from the main memories of computers to the database residing in the stable storage. Committing a sequence of steps represents irrevocable changes made to the database. In order to avoid cascaded aborts, the computational results of a transaction must be withheld until it has successfully committed. For example, if we use the two phase lock protocol to enforce the serializability of concurrency control and want to avoid cascaded aborts, then the two

phase lock protocol must be modified as follows. Instead of releasing locks after a transaction has acquired all its locks, the transaction now must not release any of its acquired writelocks until it has committed [Wood 80]. This is because a transaction can be aborted by failures before it has committed. To release a writelock before a transaction has committed would allow other transactions to use the results of a partial computation that could be invalidated by a later abort operation.

Having discussed the basic ideas of serializability theory and failure atomicity, we now review three important developments in this area: the generalization of the syntax of transactions from a single level transaction to nested transactions by Moss [Moss 81]; the investigation of using an arbitrary set of primitive transaction steps in addition to the commonly used "read step" and "write step" by Korth [Korth 83]; and the integration of serializability theory with the software engineering approach, abstract data type, by Weihl [Weihl 84]. In the early work on serializability theory, the syntax of transactions was restricted to be a single level with two primitive database operations "read" and "write". Moss [Moss 81] generalized the syntax of transactions into the form of nested transactions. The syntax of a nested transaction has the structure of a tree, where the root is called the top level transaction and where nodes are called sub-transactions. The leaves of the tree represent transaction steps. Lower level sub-transactions are invoked by higher level ones and sub-transactions can be executed in parallel. Moss also developed a lock passing scheme to carry out the two phase lock protocol in the context of a nested transaction. This scheme states that a sub-transaction can acquire new locks, but it cannot release locks to other transactions. At the end of execution, a sub-transaction must pass its locks either to another sub-transaction which waits for the locks or to its parent. Eventually, all the locks will be held by the top level transaction, which can release its locks only if the entire nested transaction has acquired all its locks. In other words, a nested transaction, as a whole, follows the two phase lock protocol: releasing locks only after acquiring all the locks which the nested transaction needs. Within the same nested transaction, sub-transactions can pass locks to each other when they share data objects.

Nested transactions have three clear advantages over single level transactions. First, the structure of nested transactions is similar to high level languages which support the nesting of sub-programs. This makes transactions easier to write. Second, a higher degree of concurrency becomes possible because sub-transactions can be executed in parallel. Finally, the management of failure recovery also becomes more flexible. When a sub-transaction aborts, all its descendant sub-transactions must also be aborted. This is because the descendant sub-transactions use the results passed by their parent to perform computations on behalf of their parent and the abortion of their parent invalidates their computations. However, the parent of

an aborted sub-transaction has the option either to abort or to continue the computation. For example, if a parent invokes a sub-transaction to obtain some resource from node A in a network and this sub-transaction aborts, the parent has the option either to abort the computation or to invoke another sub-transaction to get the resource from node B. At the end of the nested transaction, all the executed and non-aborted steps represent the computation performed by a nested transaction under a serializable schedule. These steps must be either committed or aborted as an indivisible unit to satisfy the failure atomicity requirement. In order to avoid cascaded aborts, a nested transaction, like single level transactions, cannot release any writelock to other transactions until it has successfully committed. Although Moss investigated the use of nested transactions, it was Lynch [Lynch 83a], Beeri, Bernstein, Goodman and Lai [Beeri 83] who extended the transaction model of serializability theory to include nested transactions.

We now turn to the subject of database primitive steps in the development of serializability theory. The most commonly used set of primitive steps consists of two steps: a "read step" and a "write step". Korth [Korth 83] investigated the possibility of using an arbitrary set of primitive steps. He assumed that for each primitive database step, there is an associated lock mode. For example, we have "read lock" and "write lock" for "read step" and "write step" respectively. He studied the compatibility between lock modes. Two locks are said to be compatible if two steps can simultaneously hold their locks on the same data object and yet the resulting schedule is still serializable. For example, "read locks" are compatible, because under a serializable schedule when a transaction holds a read lock on a data object, the steps of other transactions can still put their read locks on the same data object. Korth [Korth 83] showed that the "permissiveness of a compatibility function" is determined by the commutativity of the steps in question. This means that the increase in concurrency due to the system permitting data objects to be shared by different steps is determined by the commutativity of these steps. Unfortunately, except for the read step, it is difficult to find general purpose primitive steps that commute. For example, unconditional algebraic add steps commute. But they are not often useful, because the values of a data object are often bounded. On the other hand, the conditional add steps are useful but do not commute. For example, suppose that integer data object A is initially zero with consistency constraint "$0 \leq A \leq 100$". Step t_1 is "If $A > 0$, then $A := A - 1$" and step t_2 is "If $A < 100$, then $A := A + 1$". The sequences of steps $\{t_1, t_2\}$ and $\{t_2, t_1\}$ give different values to A and thus t_1 and t_2 do not commute. In order to be commutative, we must have $t_1(t_2(A)) = t_2(t_1(A)$ for every state of A.

Due to the scarcity of useful general purpose commutative steps, the "read step" and the "write step" have become the most popular set of primitive steps for general purpose applications. On the other hand, the

identification of commutative steps can be used to good advantage in specific applications. For example, if we have data objects in the form of sets, we can improve concurrency by utilizing the fact that the union operation is commutative.

A recent development in serializability theory by Weihl [Weihl 84] is to integrate serializability theory and failure atomicity with the abstract data type approach. When an abstract data type is embedded with mechanisms which ensure serializability and failure atomicity in concurrent executions, Weihl calls it an *atomicity data type.*

The computational models of abstract data type and of serializability theory are different. An abstract data type is a computational model developed in the context of time sharing operation systems. It consists of a set of *objects* and a set of *procedures* that represent the operations on the set of objects. The objects of an abstract data type can be either data objects or abstract data types. For example, the abstract data type "List-of-Computers-in-The-Network" consists of a list of elements and a set of operations such as "insert", "delete" and "search". An element in this list is the abstract data type "Computer-i's-Resource-for-Network-Level-Sharing", which consists of data records and operations for reading and updating the record.

Serializability theory and failure atomicity have been developed in the context of database systems. The model of computation in this case is a transaction system: a set of transactions sharing a common database. Since the models of computation are different, mechanisms for concurrency control and failure recovery in database systems might not be appropriate for atomic data types. To ensure failure atomicity, Weihl required that every "activity" (transaction) must not issue commit commands until the execution had ended. Transactions satisfying this requirement are said to be "well formed" by Weihl and his theory dealed with only well formed transactions. Having adopted the requirement of failure atomicity, it leaves open the problem of how best to ensure the serializability by *local* concurrency control mechanisms embedded in atomic data types.

The two best known types of concurrency control mechanisms are the time stamp based methods and variations of the two phase lock protocol. The typical time stamp method requires each transaction to get a time stamp from a centralized system counter. The local scheduler of each node will serve the requests of transactions according to the order of their time stamps. For example, suppose that at node A there are two requests from transactions T_x and T_z with time stamps 1 and 5 respectively. Transactions T_x and T_z will be processed according to their time stamp order 1 and 5. However, if transaction T_y with time stamp 2 arrives

at node X during the execution of transaction T_x, then transaction T_x will be aborted. Transactions T_y and T_z will be processed according to their time stamp order 2 and 5. If out of order requests are infrequent, a reasonable performance can be achieved by the time stamp method. The behavior of time stamp based methods is characterized by Weihl as *static atomicity*, where "static" refers to the pre-determined total ordering of time stamps which is used for concurrency control.

Alternatively, one can use two phase lock based protocols as embedded local concurrency control mechanisms. For example, each operation of an atomic data type can be viewed as a sub-transaction of a nested transaction. We can then follow Moss' lock passing scheme to ensure the serializability of concurrency control. The behavior of two phase lock based protocols is characterized by Weihl as *dynamic atomicity*. The word "dynamic" refers to the fact that the serial schedule which is equivalent to the realized serializable schedule is not pre-determined. In addition, Weihl also investigated the behavior of the hybrid case of time stamp and two phase lock protocols, whose behavior is characterized as *hybrid atomicity*. Weihl concluded that all three types of atomicity are "optimal" in the sense that none of the three is strictly inferior to the others in terms of concurrency. To enhance the concurrency of atomic data types, Weihl also investigated methods that support the use of *user specified* commutative operations in the context of atomic data types.

In summary, the classical works on concurrency control and failure recovery are serializability theory and failure atomicity. Serializable schedules create a virtual "executing alone" environment which is sufficient for transactions to execute consistently and correctly. At the end of the execution of a transaction, the failure atomicity rule either non-divisibly transfers the results of the computation to the database or aborts the transaction and re-starts it later. In this way, serializability theory and failure atomicity ensure the consistency and correctness of concurrency control in the face of clean and soft system failures. However, the concurrency offered by serializability theory and failure atomicity is quite limited, because a transaction must withhold all the results of its computation until it has completed its computation and successfully committed. Otherwise, one must be prepared to deal with the highly undesirable event of cascaded aborts. The commonly used set of primitive steps consists of the "read step" and "write step". Due to the scarcity of useful general purpose commutative steps, except in some specific applications, it is difficult to find a richer set of primitive steps which would significantly enhance the concurrency of serializable schedules. Nonetheless, the development of serializability theory represents a milestone in the history of distributed computer systems, because this theory provides a theoretical foundation upon which *provably* consistent and correct concurrency control and failure recovery mechanisms can be designed.

1.1.1.3 Non-Serializable Concurrency Control and Failure Recovery Theories

The limited concurrency offered by serializability theory has prompted researchers to investigate non-serializable concurrency control. The fundamental difference between serializable and non-serializable concurrency control is that the latter accepts schedules which are *not* equivalent to any serial schedule but are both consistent and correct. Kung and Papadimitriou [Kung 79] showed that serializability theory is optimal when only transaction syntax information is used for scheduling. This means that we must use information in addition to the syntax information of transactions in order to obtain a higher degree of concurrency than that which is offered by serializability theory. Some researchers have focused on the use of transaction semantic information in non-serializable concurrency control. For example, Molina [Molina 83] proposed the concept of *semantically consistent* schedules. In his approach, transactions are classified into two classes. One of the classes needs a consistent view of the database while the other class does not. A schedule is said to be *semantically consistent* if under this schedule all the transactions that require a consistent view will have a consistent view. He also developed algorithms that produce semantically consistent schedules. However, he has not formalized the concept of "*does not need a consistent view*", nor has he described how this concept is related to the pre- and post-conditions of transactions. It is the users' responsibility to prove that schedules resulting from their semantic information specifications are both consistent and correct for their specific applications.

Another theoretical development was Lynch's [Lynch 83b] concept of *multi-level atomicity* for the utilization of transaction semantic information in concurrency control. In this approach, transactions are first grouped into different *classes* according to the properties of their computations. The permissible ways in which the steps of a transaction can be interleaved with those of others are specified by the *break points* in between transaction steps. The specification of break points is relative to the classes of transactions. For example, when there are two classes of transactions C_1 and C_2, there will be three sets of break point specifications. One set for transactions within class C_1, one set for transactions within class C_2 and one set for transactions belonging to different classes C_1 and C_2. Lynch also developed a hierarchical form for specifying breaking points. A schedule that satisfies such specification is said to be *multi-level atomic*. Lynch developed algorithms that produce multi-level atomic schedules. It is, however, the responsibility of users to prove that their break point specifications produce multi-level atomic schedules that are both consistent and correct for their specific applications.

There have also been some experimental works that give some interesting examples in non-serializable concurrency control. Allchin [Allchin 82] developed the concept of *end of transaction serializability*. Schwarz [Schwarz 82] used a concept called *transaction dependency relations* to utilize transaction semantic information in the construction of shared atomic data types. The focus of their work is similar to that of Weihl – the integration of the abstract data type approach with distributed concurrency control and failure recovery mechanisms – however, they do not restrict themselves to serializability theory. Allchin and Schwarz gave several interesting examples of non-serializable concurrency control, however, there has not been a formal proof of the consistency and correctness of non-serializable concurrency control based on the concepts of either "end of transaction serializability" or "dependency relations".

Currently, in the Computer Science Department of Carnegie-Mellon University, Tokuda, Clark and Locke [Tokuda 85] are developing a decentralized operating system ArchOS, which is an operating system of the Archons decentralized computer system project [Jensen 83, Jensen 84]. An important feature of ArchOS is that it supports transactions at the kernel level. In fact, all the ArchOS system primitives employ transactions. The focus of their research effort in the area of transaction facility is on the development of a new model of computation called Arobjects. This model integrates our theory with a distributed abstract data type approach. Since their work involves both the application and extension of the results developed in this thesis, we will comment on their work in Section 8.2, Future Work. Readers who are interested in the use of our theory in the development of a distributed operating system should read their work [Tokuda 85], which contains detailed examples and considerations of system development that will not be addressed in this thesis.

1.1.2 The Main Contribution of This Thesis

Recently, non-serializable concurrency control has become the frontier in the investigation of concurrency control. Existing experimental and theoretical investigations have provided evidence that the semantic information of transactions can be utilized to good advantage in some specific applications. However, at this time, no other non-serializable concurrency control study has provided general non-serializable concurrency control rules that have been proven to be consistent, correct and modular, nor has there been a general theory that coherently addresses the relationship among the concepts of consistency, correctness, modularity and optimality in non-serializable concurrency control and failure recovery. The main contribution of this thesis is to provide such a theory and to deliver provably consistent, correct and optimal *modular* concurrency control and failure recovery rules.

Our theory of modular concurrency control and failure recovery is a generalization of nested transactions, serializability theory and failure atomicity. The essential difference between our work and that of Molina and of Lynch is our emphasis on the *modularity* of concurrency control and failure recovery management.

Based upon our experience with centralized operating systems, we can expect a distributed operating system to be very complex and to be written and modified by many different programmers over a period of years. Thus, it is important to develop scheduling and recovery rules that support the modular development of system and application software. Intuitively, a scheduling rule is *modular* if it allows a programmer to write, modify and schedule his transaction independently of the rest of the transactions in the system. A recovery rule is *modular* if it allows a programmer to specify the conditions for committing and aborting the transaction independently of the rest of the transactions in the system. For example, serializability theory is a modular scheduling rule, and failure atomicity is a modular recovery rule. This is because both of them can be enforced by a programmer who knows only the details of his transaction. For example, a programmer can use the two phase lock protocol to schedule his transaction and write down the commit or abort conditions without any knowledge of the rest of the transactions in the system.

In this thesis, all our proposed scheduling and recovery rules are proved to be consistent, correct and modular. That is, users of our rules need not know the semantics of others' transactions. As long as everyone can ensure that his own transaction is consistent and correct when executing alone and that he uses our rules accordingly, the consistency and correctness of system concurrency control will be ensured in the face of clean and soft failures. This is, however, not the case with either Molina's or Lynch's approaches. Their methods examine the details of a given set of transactions and then determine how the steps of one transaction can be interleaved with those of others. As a result, the modification of an existing transaction or the addition of a new transaction could invalidate the scheduling of existing transactions. However, when problems are specific and static, their methods can be utilized to produce a degree of concurrency higher than that which is provided by our approach. In the next section, we give an informal overview of our theory.

1.2 Overview of This Thesis

We have developed a formal theory of modular scheduling rules and modular failure recovery rules, which provides us with optimal modular scheduling and recovery rules. Furthermore, this new theory is a generalization of serializability theory, nested transactions and failure atomicity. In Section 1.2.1, we describe

the main characteristics of our approach from a programmer's point of view. In Section 1.2.2, we review the main concepts of this theory and in Section 1.2.3 we discuss the main results.

1.2.1 A Programmer's View

From a programmer's point of view, under serializability theory and failure atomicity each transaction is an atomic unit of operations and the database is a non-divided unit of data objects. In our approach, both the database and transactions are divided as follows.

1. The database is partitioned into consistency preserving *atomic data sets*. As long as the consistency constraints of each atomic data set are satisfied individually, the consistency of the database is maintained.

2. Each transaction is independently partitioned into a partially ordered set of correctness preserving *elementary transactions*. As long as the post-condition of each executed elementary transaction is individually satisfied, the post-conditions of the partitioned transaction (called a *compound transaction*) are also satisfied.

3. The steps of each elementary transaction are partitioned into transaction ADS segments, each of which is the set of all the steps in an elementary transaction accessing the same atomic data set. If the steps of a transaction ADS segment are not interleaved with that of any other transaction ADS segments, then this segment is an *atomic commit segment*. Interleaved transaction ADS segments are taken as a single atomic commit segment.

Having divided the database and transactions, we then employ some concurrency control protocol to ensure that elementary transactions are executed serializably on a (atomic data) set by set basis. In addition, atomic commit segments of a transaction are committed in the order they are executed. The computational result of an atomic commit segment is withheld until it has successfully committed. One way to implement this is to associate the two phase lock protocol with each atomic commit segment. In addition, the writelocks are withheld until the atomic commit segment has been successfully committed. As a result of this approach, the following is true despite clean and soft system failures:

1. The database consistency constraints will be satisfied.

2. The post-condition of each transaction will be satisfied.

3. There will be no cascaded aborts.

4. This approach provides at least as much concurrency as any other consistent and correct modular concurrency control and failure recovery method.

The concept of modularity in the context of concurrency control and failure recovery needs some explanation. Intuitively, a modular concurrency control method is one which allows a programmer to write, modify and schedule his transaction independently of other transactions. A failure recovery method is modular if it allows a programmer to write down the commit/abort conditions independently of other transactions. For example, serializability theory is a modular scheduling method, because a programmer can write, modify and then schedule (e.g. using two phase lock) his transaction independently of other transactions in the system. Failure atomicity is a modular recovery method, because one can enforce it on a transaction by writing down the commit/abort conditions independently of other transactions.

To illustrate our approach, let us consider the following example. Suppose that transaction Get-A-and-B needs one unit of some resource at node A and another unit at node B. If it cannot have both, then it would rather have none, since the task cannot be run with only one of the resources. The resource heap at each node is modelled as an atomic data set with consistency constraint "$0 \leq$ HeapSize $\leq$ MaxSize". In order not to block the resource allocation activity at any node, we structure the transaction as a compound transaction Get-A-and-B, which is illustrated in Table 1-1.

This example illustrates the following four characteristics of our approach. First, the database is partitioned into two consistency preserving atomic data sets: Heap A and Heap B. Second, the compound transaction Get-A-and-B consists of four correctness preserving elementary transactions, which co-operatively carry out the task of the compound transaction by passing information to each other via atomic variables *Obtain-A* and *Obtain-B*. As long as the post-conditions of the executed elementary transactions are individually satisfied, the post-conditions of the compound transaction Get-A-and-B are satisfied. Although in this example each of the elementary transactions is a single level transaction, an elementary transaction can have the form of a nested transaction. Third, each elementary transaction has one transaction ADS segment which is also an atomic commit segment. These atomic commit segments are committed in the order of their execution. Note that this commit protocol violates the requirement of failure atomicity: all executed steps of a transaction must be committed as a non-divisible unit. Fourth, the locking protocol used by this transaction only ensures that each elementary transaction is executed serializably with respect to each atomic data set. Global serializability of concurrency control is not enforced. For example, suppose that we have another compound transaction Get-A-or-B, which is illustrated in Table 1-2. Transaction Get-A-or-B tries to get one unit of resource from node A and one unit of resource from node B. If it cannot get both, then it keeps whatever it has obtained.

```
CompoundTransaction Get-A-and-B;
AtomicVariable Obtain-A, Obtain-B : Boolean;
BeginSerial
  BeginParallel
    ElementaryTransaction Get-A;
    BeginSerial
      Writelock Heap A;
      Take a unit from A if available
      and indicate the result in  atomic variable Obtain-A.
      Commit and unlock Heap A;
    EndSerial;

    ElementaryTransaction Get-B;
    BeginSerial
      WriteLock Heap B;
      Take a unit from B if available
      and indicate the result in atomic variable Obtain-B;
      Commit and unlock Heap B;
    EndSerial;
  EndParallel;

  BeginParallel;
    ElementaryTransaction Put-Back-A;
    BeginSerial
      If Obtain-A and not Obtain-B
      then BeginSerial
             WriteLock Heap A;
             Put-Back the unit of A;
             Commit and unlock Heap A;
           EndSerial;
    EndSerial;

    ElementaryTransaction Put-Back-B;
    BeginSerial;
      If Obtain-B and not Obtain-A
      then BeginSerial
             WriteLock Heap B;
             Put-Back the unit of B;
             Commit and unlock Heap B;
           EndSerial;
    EndSerial;
  EndParallel;
EndSerial.
```

Table 1-1: Compound Transaction Get-A-and-B

Suppose that we execute transactions Get-A-and-B and Get-A-or-B with Heap A and Heap B with initial values of one. It is possible that transaction Get-A-and-B gets the only unit of A first while transaction Get-A-or-B gets the only unit of B first. As a result, transaction Get-A-and-B will put back the obtained unit

```
CompoundTransaction Get-A-or-B;
BeginParallel
   ElementaryTransaction Get-A;
   BeginSerial
      Writelock Heap A;
      Take a unit from A if available;
      Commit and unlock Heap A;
   EndSerial;

   ElementaryTransaction Get-B;
   BeginSerial
      WriteLock Heap B;
      Take a unit from B if available;
      Commit and unlock Heap B;
   EndSerial;
EndParallel.
```

Table 1-2: Compound Transaction Get-A-or-B

to A while transaction Get-A-or-B will keep the unit obtained from B. The result of this execution is *not* equivalent to executing these two transactions serially. When these two transactions are executed serially, one of them will get both units.

Despite the violation of both serializability theory and failure atomicity, the consistency and correctness of concurrency control will be preserved in the face of clean and soft failures. That is, the sizes of Heap A and Heap B will be positive and within their bounds, transaction Get-A-and-B will either get both units or nothing and transaction Get-A-or-B will get both units, one of the two units or nothing. For example, if compound transaction Get-A-and-B gets one of the two units, commits and then fails later, it can resume its execution consistently and correctly later by either returning the obtained unit or getting another unit. Furthermore, since elementary transaction Get-A (Get-B) leaves the database in a consistent state, transaction Get-A-or-B will execute consistently and correctly despite the failure of Get-A-and-B.

The consistency and correctness of compound transactions like Get-A-and-B and Get-A-or-B is not an isolated example but rather the result of satisfying the generalized setwise serializable scheduling rule and the failure safe rule developed in this thesis. Serializability theory and failure atomicity are shown to be special cases of them.

From an application point of view, the partition of the database and of transactions increases the system

concurrency and reduces the probability of deadlocks. For example, heaps A and B are assumed to be heavily shared data objects, it is generally a good practice to minimize the duration of locking on such data objects. Compound transaction Get-A-and-B is written with this objective in mind. We accomplish this objective by dividing possible operations on atomic data set Heap A (Heap B) into two elementary transactions: Get-A (Get-B) and Put-Back-A (Put-Back-B), so that Heap A (Heap B) is locked only when it is directly manipulated. If Get-A-and-B is coded as a nested transaction and if a unit of A (B) has been obtained, the lock on Heap A (Heap B) must be kept. Keeping the lock on Heap A (Heap B) in one node of the network, the nested transaction must obtain the lock on Heap B (Heap A) in another node and determine whether it can get units of both A and B and commit, or cannot get both and abort. These two locks can be released only after the nested transaction has either committed or aborted.

From a system performance point of view, it can be quite time consuming to wait for the lock on a remote data object in a network. When the nested transaction is waiting to operate on Heap B (Heap A), it blocks the resource allocation activity on Heap A (Heap B). If a nested transactions is used to perform operations on many heavily shared data objects distributed in a system, the blockage caused by the nested transaction could be both large in extent and prolonged in duration. Recall that in the Get-A-and-B example, the structure of compound transactions allows us to lock the heavily shared resource heap only when it is directly operated upon. The partition of the database into small units of atomic data sets and the partition of a transaction into several small elementary transactions not only improve the system concurrency but also reduce the probability of the transaction involving in a deadlock. Therefore, the development of compound transactions may have important implications in real time control systems, in which an event can trigger tasks that require resources from many distributed system elements such as signal processors, communication bandwidth, and device controllers.

1.2.2 Basic Concepts

The four basic concepts concerning concurrency control and failure recovery addressed in this work are consistency, correctness, modularity and optimality.

1.2.2.1 Consistency and Correctness

In concurrency control, two properties of transactions are of fundamental importance: consistency and correctness. By "consistent", we mean the satisfaction of the database consistency constraints, and by "correctness" we mean the satisfaction of the transaction post-conditions. In failure recovery, an important concept is "clean and soft" failures [Bernstein 83]. A system failure is said to be clean and soft, if its effect can be modelled by a network of computers, some of which stop running and lose the content of their main memories. However, the database, including the portion of the database residing in the stable storage used by the failed computers, is not affected by the failures. We assume that each transaction is consistent and correct when executing alone and that all failures are clean and soft. Under these assumptions, a schedule is said to be consistent and correct if it executes each transaction consistently and correctly. A failure recovery rule is said to be consistent and correct if the committed results of executing a set of transactions in the face of system failures is equivalent to the results of an execution produced by a consistent and correct schedule of the same set of transactions when there is no failure.

We have mentioned that serializable schedules are consistent and correct. However, serializable schedules constitute only a subset of all the possible consistent and correct schedules. In this work, we study consistent and correct scheduling rules whose associated schedules are supersets of the serializable schedules. That is, these rules permit a higher degree of concurrency by including non-serializable schedules. Under non-serializable schedules, transactions cannot be regarded as executing alone. The consistency and correctness of such scheduling rules is not obvious and must be proved.

In this work, we also study consistent and correct failure recovery rules which provide sufficient conditions upon which the consistency and correctness of the concurrency control is preserved in the face of *clean and soft* system failures. The recovery rule associated with serializability theory is known as *failure atomicity*, which specifies that a transaction either commits all its executed steps at the end of its execution or aborts these executed steps. We have developed a consistent and correct recovery rule that specifies the conditions under which the computations performed by a transaction can be committed part by part. In other words, our recovery rule performs distributed "check-points" so that transactions not affected by failures execute consistently and correctly and transactions interrupted by failures can resume their computations consistently and correctly.

1.2.2.2 Modularity and Optimality

Based upon our experience with centralized operating systems, we can expect a distributed operating system to be very complex, and to be written and modified by many different programmers over a period of years. Thus, we must develop scheduling rules supporting the modular development of system and application software. Intuitively, a scheduling rule is modular if it allows a programmer to write, modify and schedule his transaction independently of any knowledge of the rest of the transactions in the system. A recovery rule is modular if it allows a programmer to specify the conditions for committing and aborting the transaction independently of the rest of the transactions in the system. For example, serializability theory is a modular scheduling rule and failure atomicity is a modular recovery rule. This is because both of them can be enforced by a programmer who knows only his transaction. For example, he can use the two phase lock protocol to schedule his transaction and write down the commit or abort conditions without any knowledge of the rest of the transactions in the system.

The optimality of scheduling rules is modelled by set containment. A scheduling rule is optimal under some given conditions if the set of schedules permitted by this rule is the superset of the permissible schedules of any other rule under the same conditions. The optimality of recovery rules is modelled in a similar fashion.

Both modularity and optimality are characterized by the way they utilize the information regarding the transaction system and the database. We assume that transactions are scheduled one at a time. The information available to scheduling and recovery rules can be classified as follows:

1. The consistency constraints of the database.

2. The syntactic definition of transactions, such as the definition of nested transactions. Note that the syntactic definition includes the definition of the set of primitive steps used by the transactions, e.g., "read step" and "write step".

3. The local semantic information: the description of the computation performed by the transaction which is being scheduled.

4. The global semantic information: the description of the computation performed by a *set* of transactions. When a scheduling rule utilizes the semantic information of any transaction other than the one being scheduled, we say that this rule uses global semantic information.

When a programmer writes or modifies a transaction, he must know the consistency constraints to verify the consistency of his transaction. In addition, he must know the syntax of transactions and understand the

semantics of his transaction to write down the code correctly. In other words, when he is ready to schedule his transaction and to write down the conditions for aborting or committing his transaction, he must use the first three types of information. Thus, the designer of modular scheduling and recovery rules should exploit the use of such information.

On the other hand, the designer of modular scheduling and recovery rules should avoid the use of global semantic information. This is because when one uses the global semantic information, the modification of one transaction could force the re-scheduling of other transactions. To illustrate the problems associated with using global semantic information, let us consider the following example. Suppose that we have a database consisting of two variables A and B with consistency constraint "A + B = 100". There are two transfer transactions: $T_1 = \{t_{1,1}: A := A - 1; t_{1,2}: B := B + 1\}$ and $T_2 = \{t_{2,1}: B := B - 2; t_{2,2}: A := A + 2\}$, where the symbol "$t_{i,j}$" denotes the step j of transaction i. In addition, there is another implementation of transaction T_2: $T_2^* = \{t_{2,1}^*: B := B - 2; t_{2,2}^*: A := 100 - B\}$. Note that transaction T_2^*, like T_2, transfers two units from B to A and preserves the consistency constraint "A + B = 100" when executing alone. Consider the following two non-serializable schedules: $S_1 = \{ t_{1,1}; t_{2,1}; t_{2,2}; t_{1,2} \}$ and $S_2 = \{t_{1,1}; t_{2,1}^*; t_{2,2}^*; t_{1,2} \}$. Schedule S_2 is the same as S_1, except that the steps of T_2 are replaced by the steps of T_2^*. It is easy to check that S_1 is consistent and correct, while S_2 is inconsistent. In Table 1-3, there are two examples of executing schedules S_1 and S_2.

An Execution of Schedule S_1

A (50)	B (50)
$t_{1,1}$: A := A - 1; (49)	
	$t_{2,1}$: B := B - 2; (48)
$t_{2,2}$: A := A + 2. (51)	
	$t_{1,2}$: B := B + 1. (49)

An Execution of Schedule S_2

A (50)	B (50)
$t_{1,1}$: A := A - 1: (49)	
	$t_{2,1}^*$: B := B - 2; (48)
$t_{2,2}^*$: A := 100 - B. (52)	
	$t_{1,2}$: B := B + 1. (49)

Table 1-3: Executions of Schedules S_1 and S_2

Note that at the end of executing S_2, the sum of A and B is 101, an inconsistent result. Using global semantic information about both T_1, T_2 and T_2^*, a scheduling rule is able to accept S_1 and reject S_2. However, this rule must know the *interactions* among the detailed codings of these transactions. The knowledge of the database consistency constraints and the post-conditions of these transactions is insufficient. This is demonstrated by the fact that S_2 is inconsistent, although the post-conditions of T_2 and T_2^* are identical.

When a scheduling rule relies upon the details of several transactions to schedule each of the transactions, a slight modification in one transaction could invalidate the scheduling of some other transactions. This could cause serious difficulties in the development of a general atomic transaction facility, since new transactions are constantly written and existing ones are frequently updated. In summary, a modular rule only uses the first three types of information, and the optimal modular rule makes the best use of them. Finally, we want to point out that when problems are specific and static, global semantic information could be utilized to good advantage [Allchin 82, Schwarz 82, Molina 83, Lynch 83b]. Such an approach is, however, outside the scope of this work.

1.2.3 Main Contributions

There are four main contributions in this thesis. The first is the formal model of modular scheduling rules. The second is the *setwise serializable* scheduling rule, which is a scheduling rule which produces non-serializable schedules. This rule is proven to be both consistent, correct and modular; and it is the optimal application independent scheduling rule. The third is a new transaction structure: *compound transactions* and their associated *generalized setwise serializable* scheduling rules. These rules form a complete class within the set of all the possible consistent and correct modular scheduling rules. This means that for any consistent and correct modular scheduling rule, there always exists a generalized setwise serializable scheduling rule that provides at least as much concurrency in scheduling. The fourth main contribution is the failure recovery rule called the *failure safe* rule. This rule is the optimal failure recovery rule for compound transactions scheduled by generalized setwise serializable scheduling rules.

1.2.3.1 A Model of Modular Scheduling Rules

Our first main contribution is the development of a formal model of modular scheduling rules. This model provides us with a formal definition of the intuitive concept of "modularity" in the context of concurrency control, and permits us to have a unified treatment of different modular scheduling methods.

Intuitively, a scheduling rule specifies how the steps of a transaction can be interleaved with those of other transactions. In our model, a scheduling rule specifies the permissible ways of interleaving transaction steps by partitioning the steps of each transaction into equivalent classes called *atomic step segments*. Schedules satisfy the specification of a given scheduling rule by interleaving the atomic step segments of each transaction serializably with the atomic step segments of other transactions. For example, serializability theory is a particular scheduling rule which takes all the steps of a transaction as a single atomic step segment. Serializable schedules are the set of schedules that satisfy this rule, because the atomic step segments specified by serializability theory are interleaved serializably in serializable schedules. A scheduling rule is said to be consistent and correct if and only if all the schedules satisfying this rule are consistent and correct. For example, serializability theory is consistent and correct, because all the serializable schedules are consistent and correct.

We now introduce the concept of modularity in concurrency control. A scheduling rule is said to be modular if this rule partitions each transaction independently of other transactions in the system. For example, serializability theory "partitions" the steps of each transaction independently of the rest of transactions in the system. Thus, serializability theory is a modular scheduling rule. When a transaction is being partitioned by a modular rule, the only information regarding *other* transactions in the system is that they are supposed to be consistent, correct and written in the syntax agreed upon. Nonetheless, given any arbitrary set of such independently partitioned transactions, the modular scheduling rule that produces such partitions must guarantee the consistency and correctness of all the schedules in which atomic step segments of one transaction are interleaved serializably with those of other transactions.

The degree of concurrency produced by a scheduling rule is determined by the degree of interleaving the rule permits, thus finer partitions lead to a larger set of permissible schedules and a higher degree of concurrency. The highest degree of concurrency is reached when the partition is so fine that each step is an atomic step segment, while the lowest degree of concurrency is reached when the entire set of steps in a transaction is taken as a single atomic step segment. However, a rule can only partition a transaction to the extent that it can still guarantee the consistency and correctness of the resulting schedules with respect to the available information. For example, with only transaction syntax information, there is little a scheduling rule can do except take the entire transaction as a single atomic step segment. Serializability theory partitions each transaction in this way. Kung and Papadimitriou [Kung 79] proved that serializability theory is optimal when only transaction syntax information is used for scheduling. We have investigated how to utilize other types of information that are also available to modular scheduling rules.

Given a set of scheduling rules, we are interested in knowing which one provides a higher degree of concurrency. To this end, we model the intuitive concept: "the degree of interleavings permitted by a scheduling rule" by the formal concept of set containment. If the set of schedules that satisfies scheduling rule R_1 is a subset of that which satisfies R_2, then we say that R_2 is more concurrent than R_1. We consider that a consistent and correct modular scheduling rule is optimal under some conditions if this rule provides at least as much concurrency as any other consistent and correct modular scheduling rule under the same conditions. For example, serializability theory is optimal under the condition that only transaction syntax information is used for scheduling [Kung 79].

The optimality of scheduling rules is relatively easy to investigate when information used for scheduling is not too rich, such as is the case when only transaction syntax information is used. We will also identify the optimal scheduling rule when the consistency constraints of the database and transaction syntax information are both used for scheduling. However, when the semantic information of transactions is also available for scheduling, the identification of the optimal scheduling rule(s) becomes extremely difficult. In this case, we settle for a less ambitious goal: completeness. We will develop a family of modular scheduling rules for users. This family of rules is complete within the set of consistent and correct modular scheduling rules. This means that given any consistent and correct scheduling rule R^*, a user can always find a rule R in this family such that R provides at least as much concurrency as R^*.

1.2.3.2 The Setwise Serializable Scheduling Rule

Our second main contribution is the development of the optimal scheduling rule when both syntax information and the database consistency constraints are utilized. This rule is called the *setwise serializable* scheduling rule. To understand this rule, we must first introduce the concepts of *atomic data sets* and the *consistency preserving partition* of the database. To perform a *consistency preserving partition* of the database, we partition data objects into disjoint sets in such a way that the consistency of each set can be maintained independently of the others. Furthermore, the conjunction of the consistency constraints of each of the sets is equivalent to the database consistency constraints. When a database is partitioned in this way, each of the sets in the partition is called an *atomic data set* (ADS).

Intuitively, an atomic data set is a collection of data objects with a set of consistency constraints that can be satisfied independently of other atomic data sets. For example, in a distributed computer system a job queue at a computer waiting for execution is an atomic data set, the local directory to a computer's files is an atomic

data set and each user's mailbox is also an atomic data set. This is because the consistency of each of these entities is defined by a set of consistency constraints that can be satisfied independently of others. As a more detailed example, suppose that we have only two data objects: a document queue (Q_d) and a printer queue (Q_p) distributed in two different nodes in a local network. To bound the queue size, we specify that "$0 \leq Q_d \leq Max_d$" and "$0 \leq Q_p \leq Max_p$", where Max_d and Max_p are the maximal sizes of Q_d and Q_p respectively. These two inequalities are now our database consistency constraints. To model the "producer-consumer" relation between the document and printer queues, we specify the post-condition for the file transfer transaction: the sum of these two queues is not altered by the file transfer operation, "$Q_d + Q_p = Q_d^* + Q_p^*$" where Q_i and Q_i^*, $i = d, p$, are the states input to and output from the file transfer transaction. In this example, we have two atomic data sets Q_d and Q_p together with their consistency constraints "$0 \leq Q_d \leq Max_d$" and "$0 \leq Q_p \leq Max_p$" respectively. It is important to note that inter-related data objects can be in different atomic data sets, as long as the consistency of an atomic data set can be maintained independently of other atomic data sets.

We are now in a position to define the setwise serializable scheduling rule. Recall that a modular scheduling rule partitions the steps of a transaction into atomic step segments that must be executed serializably by the schedules associated with the rule. Given a transaction, the setwise serializable scheduling rule partitions the transaction into atomic step segments in the form of *transaction ADS segments*. A transaction ADS segment is defined as all the steps in a transaction that access data objects in the same atomic data set. For example, the file transfer transaction above has two sets of steps: one set operates upon the document queue while the other set operates upon the printer queue. Since each queue is an atomic data set, each of the two sets of steps is a transaction ADS segment. We have proved that the setwise serializable scheduling rule is consistent, correct and modular. In this work, a modular scheduling rule that does not utilize any transaction semantic information is referred to as an *application independent* rule. We have also proved that the setwise serializable scheduling rule is the optimal application independent rule. Owing to the key role played by the consistency preserving partition, we will present a more detailed example to illustrate this concept in Chapter 7, An Example of Application.

1.2.3.3 Compound Transactions

The third main contribution of this thesis is the development of a new transaction structure: *compound transactions*, and its associated scheduling rules called *generalized setwise serializable* scheduling rules. The structure of compound transactions is designed to use the semantic information of a given transaction by partitioning the steps of a transaction into a partially ordered set of *elementary transactions*.

Conceptually, the development of compound transactions is parallel to that of atomic data sets discussed in the previous section. While atomic data sets result from a consistency preserving partition of the database, elementary transactions result from a correctness preserving partition of a transaction. Given a database and its associated consistency constraints, we partition the database into consistency preserving atomic data sets. As long as the consistency of each atomic data set is satisfied individually, the consistency of the database is also satisfied. Given a transaction and its associated post-conditions, we partition the transaction into correctness preserving elementary transactions. As long as the post-condition of each elementary transaction is individually satisfied, the post-conditions of the compound transaction are also satisfied. Each elementary transaction must preserve the consistency of the database and satisfy its own post-condition when it is executed serially and in an order consistent with the partial ordering of elementary transactions in the compound transaction.

Given a compound transaction, the generalized setwise serializable scheduling rule partitions the steps of each elementary transaction into transaction ADS segments. This means that as long as we execute all the elementary transactions setwise serializably and in an order consistent with the partial orderings of elementary transactions defined by compound transactions, the concurrent executions of compound transactions will be consistent and correct.

Generally, an elementary transaction can have the structure of a nested transaction. In fact, a compound transaction becomes a nested transaction when it consists of only one elementary transaction. Elementary transactions in a compound transaction can exchange the results of their computations. When an elementary transaction is executed serializably and in an order consistent with the partial ordering of the elementary transactions in the compound transaction, it has the following two properties. First, given a consistent state of the database and the results passed from preceding elementary transactions, an elementary transaction produces another consistent state of the database and satisfies its own post-conditions. Second, the conjunction of the post-conditions of the constituent elementary transactions is equivalent to the post-conditions of the compound transactions.

Owing to these two properties, the locks acquired by an elementary transaction can be released at the end of its execution. This ability can be used not only for the enhancement of concurrency but also for the purpose of data object encapsulation and protection. We can take the abstract data type approach and let the elementary transactions be the operations of an abstract data type. Owing to the property of elementary transactions, locks on critical system data objects can be released rather than passed to users. This is important because once the locks on sensitive system objects are passed to users, the system loses the control on these objects until locks are returned to the system.

In this thesis, we prove that generalized setwise serializable scheduling rules form a complete class within the set of all the consistent and correct modular scheduling rules. From an application point of view, this means that the approach partitioning the database and of writing and scheduling compound transactions provides at least as much concurrency as any other modular concurrency method.

1.2.3.4 The Failure Safe Rule

Our last main contribution concerns the failure safe rule used in failure recovery. In serializability theory, the associated failure recovery rule is the concept of "failure atomicity", which requires all the executed steps of a transaction to be either committed or aborted. In contrast, the failure safe rule partitions a compound transaction into *atomic commit segments* as follows. Take a transaction ADS segment in each of the elementary transactions and examine it to see if its steps are interleaved with another transaction ADS segment in the same elementary transaction. If it is not interleaved with others, then it is an atomic commit segment. If it is interleaved with others, then merge these interleaved transaction ADS segments into a single atomic commit segment. At the end of this process, we have created a *partially ordered* set of atomic commit segments. In the example compound transaction Get-A-and-B, each elementary transaction is an atomic commit segment, because each elementary transaction consists of only one transaction ADS segment.

As long as each transaction commits its atomic commit segments in a sequence consistent with this partial order, the consistency and correctness of concurrency control is ensured in the face of system failures. When a failure occurs, transactions not affected by the failure continue executing consistently and correctly, while transactions affected by the failure can later resume their computation consistently and correctly. For example, the compound transaction Get-A-and-B can commit its atomic commit segments: Get-A, Get-B, Put-Back-A and Put-Back-B in an order consistent with the partial order of these segments. Suppose that after Get-A has been committed, the compound transaction fails. The database is still consistent since Get-A

is consistency preserving. Therefore, other transactions can continue executing consistently and correctly. When compound transaction Get-A-and-B resumes its execution, it can either execute Get-B to complete the task or execute Put-Back-A if B is not available. In either case, the post-condition of Get-A-and-B is satisfied. In this thesis, we prove that the failure safe rule is consistent, correct and modular. In addition, we also prove that this rule is optimal in the set of all the possible modular recovery rules that can be designed for compound transactions scheduled by generalized setwise serializable scheduling rules.

1.2.4 Summary

In this overview, we have outlined the characteristics of our approach and discussed four basic concepts and four main contributions of our theory of modular concurrency control and failure recovery. The four concepts are *consistency*, *correctness*, *modularity* and optimality. The four main contributions are a formal model of modular scheduling rules, the setwise serializable scheduling rule, the compound transactions and their associated generalized setwise serializable scheduling rules, and the failure safe rule. The formal model of modular scheduling rules permits us to have a unified treatment of different modular scheduling methods. The setwise serializable scheduling rule is the optimal application independent scheduling rule. The generalized setwise serializable scheduling rules form a complete class within the set of all the modular scheduling rules. This means that for any given modular scheduling rule, there always exists a generalized setwise serializable scheduling rule that provides at least as much concurrency. The failure safe rule is the optimal failure recovery rule for any given set of compound transactions scheduled by the generalized setwise serializable scheduling rules. Finally, our theory includes nested transactions, serializability theory and failure atomicity as special cases.

1.3 Organization of This Thesis

We begin our formal investigation in Chapter 2 by first developing a formal model of modular scheduling rules. In this chapter, we first define the basic concepts related to the database and transactions. We then introduce the concept of a modular scheduling rule. Finally, we define the important properties of modular scheduling rules such as consistency, correctness and optimality. This model provides the foundation upon which we develop our new transaction structures and modular scheduling rules. Having developed the model of modular scheduling rules, we move on to the investigation of the implications of database consistency constraints in Chapter 3. In this chapter, we develop two key concepts called a *consistency preserving* partition of the database and *atomic data sets*. These two concepts and related theorems are the basis on which we organize shared data objects in the system.

Having developed the model of modular scheduling rules and the concepts of atomic data sets and of the consistency preserving partition of the database, we develop our new non-serializable scheduling rules in Chapter 4. In this chapter, we introduce the optimal application independent scheduling rule called the *setwise serializable* scheduling rule, which utilizes the properties of atomic data sets for the purpose of concurrency control. We prove that this scheduling rule is consistent, correct and modular. Finally, we prove that this rule is the optimal application independent scheduling rule. In Chapter 5, we remove the requirement of application independence so that we can obtain a higher degree of concurrency than that which is provided by the setwise serializable scheduling rule. In this chapter, we develop a new transaction structure called *compound transactions* and their associated scheduling rules called *generalized setwise serializable* scheduling rules. We prove that these rules are consistent, correct and modular. We conclude this chapter by proving that these rules form a complete class within the set of all the possible consistent and correct modular scheduling rules. This means that for any given consistent and correct modular scheduling rules, there is a generalized setwise scheduling rule which provides at least as much concurrency.

Having studied modular scheduling rules, we investigate failure recovery rules designed for our proposed new transaction structures and scheduling rules in Chapter 6. In this chapter, we first define the concepts of consistency, correctness, modularity and optimality for failure recovery rules. Next, we define a new recovery rule called the *failure safe* rule, and we prove that this rule is consistent, correct and modular. We conclude this chapter by proving that this rule is optimal for all the possible modular failure recovery rules that can be designed for compound transactions scheduled by generalized setwise serializable scheduling rules. Chapter 7 is an example of a distributed directory system, which is used to illustrate the basic concepts of this theory. Chapter 8 presents the conclusion of this thesis and discusses the future work.

2. A Model of Modular Scheduling Rules

In this chapter, we begin our formal investigation by developing a model of modular scheduling rules. This model is the foundation upon which we develop new transaction structures and scheduling rules in later chapters. Many basic concepts in this chapter, such as "database" and "transactions", are similar to those seen in the database literature, but the concepts related to modular scheduling rules are new. Before formally defining all the concepts needed in this model, we give an overview of some of these new concepts.

Intuitively, a scheduling rule specifies how the steps of a transaction can be interleaved with those of other transactions. In our model, a scheduling rule specifies the permissible ways of interleaving transaction steps by partitioning the steps of each transaction into equivalent classes called *atomic step segments*. Schedules satisfy the specification of a given scheduling rule by interleaving the atomic step segments of each transaction serializably with the atomic step segments of other transactions. For example, serializability theory is a particular scheduling rule which takes all the steps of a transaction as a single atomic step segment. Serializable schedules are the set of schedules that satisfy this rule, because the atomic step segments specified by serializability theory are interleaved serializably in serializable schedules.

We now define some important properties of scheduling rules. A scheduling rule is said to be *modular* if this rule partitions each transaction independently of other transactions in the system. For example, serializability theory "partitions" the steps of each transaction independently of the rest of transactions in the system. Thus, serializability theory is a modular scheduling rule. A scheduling rule is said to be consistent and correct if and only if all the schedules satisfying this rule are consistent and correct. In order to discuss the optimality of scheduling rules, we need a way to compare the concurrency provided by different scheduling rules. Such a way is provided by set containment. For example, if the set of schedules satisfying a scheduling rule R_1 is a subset of that satisfying another scheduling rule R_2, then rule R_2 is said to be more concurrent than rule R_1. We consider a consistent and correct modular scheduling rule to be *optimal* under some conditions if this rule is at least as concurrent as any other consistent and correct modular scheduling rule under the same conditions. For example, serializability theory is optimal under the condition that only syntax information of transactions is used for scheduling [Kung 79].

This chapter is organized as follows. In Section 2.1, we define the basic concepts related to the concept of the database. In Section 2.2, we formalize the basic concepts regarding transactions and their schedules, and in Section 2.3, we develop our model of modular scheduling rules.

2.1 Database

A database is simply a set of shared data objects and a state of the database is a vector of the values of these shared data objects. A consistent state is a state that satisfies a given set of consistency constraints. For example, if we have a simple database consisting of two queues, Q_1 and Q_2 with consistency constraints "$0 \leq Q_1 \leq 100$" and "$0 \leq Q_2 \leq 100$", then a state is a description of the length of Q_1 and the length of Q_2. A consistent state is a state in which both the length of Q_1 and Q_2 are positive numbers and less than or equal to 100. We now formalize the concepts related to the concept of database by first defining the concept of a data object and its representation.

Definition 2.1-1: A data object, **O**, is a user defined smallest unit of data which is individually accessible and upon which synchronization can be performed (e.g. locking).

Definition 2.1-2: Associated with each data object **O**, we have a set Dom(O), the domain of **O**, consisting of all possible values taken by **O**.

Definition 2.2: Each data object is represented by triplets, <name, value, version number>. When a data object is created, its initial value is assigned to version zero of this data object, e.g. "A[0]: = 1". When the data object is updated, a new version of the object is created, and the transaction works on this new version. The version number will be incremented when the update is completed.

As an example, the step "A : = A + 1" in a transaction corresponds to the following:

```
A[v+1] := A[v];   {where v denotes the current version}
A[v+1] := A[v+1] + 1;
v   := v+1;
```

In the following discussion, when we refer to the current value of a data object **O**, we would, for simplicity, write "**O**" instead of "**O**[v]". The version number representation will be used when different versions of the values of a data object are referred to. Having defined the concept of data objects, we are now in a position to define the concept of the database.

Definition 2.3-1: The system database $D = \{O_1, O_2, \ldots, O_n\}$ is the collection of all the shared data objects in the system.

Definition 2.3-2: A state of database D is an n-tuple $Y \in \Omega = \prod_{j=1}^{n} Dom(O_j)$.

Definition 2.3-3: Associated with database D, there is a set of consistency constraints in the form of predicates on the states of database D. A consistent state of database D is an n-tuple, Y, satisfying this set of consistency constraints. This is indicated by "C(Y) = 1", where C is a Boolean function indicating whether this set of consistency constraints is satisfied by Y. For simplicity, we will also refer to this set of consistency constraints by C. The meaning of C is easily determined by the context. The set of all consistent states of D is denoted by U, where $U = \{Y \mid C(Y) = 1\}$.

2.2 Transactions and Their Schedules

Having formalized the concept of the database, we now investigate the concepts of transactions and their schedules. In its simplest form, a transaction is a sequence of transaction steps. Each step is a non-divisible operation on a data object of the database. For example, the transaction Enqueue(Q_1, *item*) = {Writelock Q_1; if $Q_1 < 100$ then put *item* at the tail of Q_1 by updating the appropriate pointers; Unlock Q_1.} is a transaction with a single *write* step. The lock and unlock pair of operations is a mechanism that makes the complex operations of enqueue appearing to be an indivisible step on data object Q_1.

There are different types of transactions. When the steps of a transaction are in the form of a sequence, this transaction is called a *single level* transaction. When the steps of a transaction are organized into a hierarchical form of nested sub-transactions, this transaction is called a *nested* transaction. Obviously, a single level transaction is a special case of a nested transaction. A nested transaction is, however, a special case of a *compound* transaction, which was informally introduced in the overview and which will be investigated in detail in Chapter 5. In this chapter, we focus on single level transactions, because they are the simplest to study first.

A transaction is said to be consistent and correct if it preserves the consistency of the database and satisfies its own post-conditions when executing alone. In this thesis, we study only consistent and correct transactions. The concurrent executions of a given set of transactions are modelled by a schedule, which is a total ordering of all the transaction steps of a given set of transactions. In addition, the steps in a schedule must be consistent with the orderings of steps in each of the transactions. For example, suppose that we have two transactions $T_1 = \{t_{1,1}, t_{1,2}\}$ and $T_2 = \{t_{2,1}, t_{2,2}\}$. The sequence of steps $\langle t_{1,1}, t_{2,1}, t_{1,2}, t_{2,2}\rangle$ is a schedule, but the sequence $\langle t_{1,2}, t_{1,1}, t_{2,1}, t_{2,2}\rangle$ is not a schedule.

Assuming that each transaction is consistent and correct when executing alone, a schedule is said to be consistent and correct if under this schedule each of the transactions is executed consistently and correctly. In the study of schedules, another important concept is called the equivalence of schedules. Two schedules are said to be equivalent if and only if they induce identical ordering of steps on each of the shared data objects. For example, let schedule $z_1 = \langle t_{1,1}(O_1), t_{2,1}(O_1), t_{1,2}(O_2), t_{2,2}(O_2)\rangle$ and schedule $z_2 = \langle t_{1,1}(O_1), t_{1,2}(O_2), t_{2,1}(O)_1), t_{2,2}(O_2)\rangle$. Schedules z_1 and z_2 are equivalent, because in both schedules the operations on data objects O_1 and O_2 can be represented as "$t_{2,1}(t_{1,1}(O_1))$" and "$t_{2,2}(t_{1,2}(O_2))$".

It is intuitively clear that if two schedules are equivalent and one is consistent and correct, then the other must be so. This intuition is formally proved as Theorem 2.1. As an example of equivalent schedules, we know that serial schedules are consistent and correct. By Theorem 2.1, all the schedules equivalent to serial schedules are also consistent and correct. The union of the set of serial schedules and its equivalent set of schedules is known as serializable schedules.

Having reviewed the concepts of transactions and their schedules, we now formalize the concept of transactions in Section 2.2.1 and the concept of schedules in Section 2.2.2.

2.2.1 Transactions

In this section, we first define the syntax of single level transactions and the concepts of pre- and post-conditions of a transaction. We then enumerate our fundamental assumptions regarding transactions. In later chapters, we will introduce generalizations to the single level transactions defined here. In the following discussion, the word "transaction" means "single level transaction" unless it is explicitly stated otherwise, such as "nested transaction" and "compound transaction".

Definition 2.4: A single level transaction T_i is a sequence of transaction steps ($t_{i,1}, t_{i,2}, \ldots t_{i,m_i}$) with the following syntax:

<SingleLevelTransaction> ::= BeginSerial <StepList> EndSerial.

<StepList> ::= Step | <StepList>;<StepList>

A transaction step is modelled as the non-divisible execution of the following instructions [Kung 79]:

$$L_{t_{i,j}} := O_{t_{i,j}}$$

$$O_{t_{i,j}} := f_{t_{i,j}}(L_{t_{i,1}}, L_{t_{i,2}}, \ldots, L_{t_{i,j}})$$

where the symbol "$t_{i,j}$" represents step j of transaction T_i; the local variable $L_{t_{i,j}}$ is used by step $t_{i,j}$ to store the value read. The symbol "$O_{t_{i,j}}$" is the data object accessed (read or written) by step $t_{i,j}$ and the symbol "$f_{t_{i,j}}$" represents the computation performed by step $t_{i,j}$.

In this model, every step reads and then writes a data object. A read step is interpreted as writing the value read back to the data object. That is, the function $f_{t_{i,j}}$ associated with a read step is the identity function. It is well known that concurrency can be improved by using a richer set of primitive steps in addition to the "read" and "write" steps. We will address this point when we discuss the issue of optimality in Section 2.3.3.

Having defined the syntax, we need to define the pre- and post-conditions of a transaction. We begin with defining the input steps and output steps in a transaction.

Definition 2.5: Let $T_i = \{t_{i,1}, \ldots, t_{i,m_i}\}$ be a transaction. Let data object O be the one accessed (read or written) by step $t_{i,j}$. Step $t_{i,j}$ is said to be an *input step* if it is the step in T_i that first accesses data object O. Step $t_{i,j}$ is said to be an *output step* if it is the step in T_i last accessing O. That is, for every data object O accessed by T_i there is an input step and an output step associated with O. Note that when there is only one step in T_i accessing O, then this step is both an input and an output step.

Since transactions operate on the shared database, they must be able to accept any consistent state as its input. That is, the values input to a transaction are assumed to satisfy the pre-conditions of the transactions as long as these values come from a consistent database state.

Definition 2.6: Let $O_{i,j}[v_b]$, j = 1 to k_i, be the set of values read by the input steps of T_i, where v_b denotes the version of a data object that is input to a transaction. Let the index set of $O_{i,j}[v_b]$, j = 1 to k_i, be I_m. The input values to T_i, $O_{i,j}[v_b]$, j = 1 to k_i, are said to satisfy the pre-condition of T_i, if and only if

$$\exists (X \in U)(\pi_{I_m}(X) = O_{i,j}[v_b], j = 1 \text{ to } k_i)$$

where π_{I_m} is the projection operator. That is, $\pi_{I_m}(X)$ is a tuple whose elements are those of X indexed by I_m.

In other words, Definition 2.6 states that a transaction must function properly if all the values input to the transaction come from a consistent state of the database. Having discussed the pre-conditions, we now turn to the subject of post-conditions.

Definition 2.7: Let $O_{i,j}[v_f]$, j = 1 to k_i, be the set of values written by the output steps of transaction T_i, where v_f denotes the version of a data object output by the transaction. The *post-condition* of transaction T_i is the specification of the output values of T_i as functions of the input values,

$$O_{i,j}[v_f] = g_{i,j}(O_{i,1}[v_b], \ldots, O_{i,k_i}[v_b]), \; j = 1 \text{ to } k_i.$$

Having defined concepts relating to the notion of a transaction, we now state our fundamental assumptions about a transaction:

Fundamental Assumptions:

- A1 *Termination*: A transaction is assumed to have a finite number of steps and assumed to terminate.
- A2 *Transaction Correctness*: A transaction is assumed to produce results that satisfy its post-condition when executing alone and when the database is initially consistent.
- A3 *Transaction Consistency*: Given a consistent state of the database, a transaction is assumed to produce a state that satisfies the consistency constraints of the database when it is executing alone.

Definition 2.8: A transaction T_i is said to be consistent and correct if and only if T_i satisfies assumptions A1, A2 and A3.

In the following discussion, we restrict our attention to consistent and correct transactions.

2.2.2 Schedules

In the previous two sections, we have formalized the basic concepts related to database and transactions. We now develop our model one step further by considering the execution of a set of transactions. To this end, we introduce the concepts of transaction systems and schedules.

Definition 2.9: A transaction system T is a finite set of transactions $\{T_1, \ldots, T_n\}$ operating upon the shared database D.

Definition 2.10: A schedule z for transaction system T is a totally ordered set of all the steps in the transaction system $T = \{T_1, ..., T_n\}$ such that the ordering of steps of T_i, i = 1 to n, in the schedule is consistent with the ordering of steps in the transaction T_i, i = 1 to n.

$$[\forall (t)((t \in z) \Leftrightarrow (t \in \cup T))] \wedge [\forall (T_i \in T) \forall (((t_{i,j}, t_{i,k} \in T_i) \wedge (t_{i,k} > t_{i,j})) ((t_{i,j}, t_{i,k} \in z) \wedge (t_{i,k} > t_{i,j}))]$$

A schedule z for transaction system T is said to be consistent if and only if the execution of T according to z preserves the consistency of the database D. This concept is formalized as follows.

Definition 2.11: Let X be the initial state of D and Y be the state at the end of executing T according to z. Schedule z is said to be consistency if and only if

$$z: X \in U \rightarrow Y \in U$$

A schedule z for a transaction system T is said to be correct if and only if the execution of transactions in T according z produces computations that satisfy their respective transaction post-conditions.

Definition 2.12: Under schedule z, let the values input to and output from transaction $T_i \in T$ be $O_{i,1}[v_b]$, ..., $O_{i,k_i}[v_b]$ and $O_{i,1}[v_f]$, ..., $O_{i,k_i}[v_f]$ respectively. Schedule z is said to be correct if and only if

$$\forall (T_i \in T)(O_{i,j}[v_f] = f_{i,j}(O_{i,1}[v_b], ..., O_{i,k_i}[v_b]), j = 1 \text{ to } k_i)$$

Having defined the basic properties of a schedule, we now consider the relations between schedules by introducing the concept of equivalent schedules. Conceptually, two schedules z and z^* for transaction system T are equivalent if for any given initial state of D the executions of T according to z and z^* yield the same sequences of values for each data object in the database and the same sequences of values for each of the local variables (states) of each transaction in T. This is formally defined by the partial ordering of steps induced by z and z^* on each of the data objects in D.

Definition 2.13: Let $O_{t_{i,j}} \equiv O_{t_{k,m}}$ denote step $t_{i,j}$ and step $t_{k,m}$ read or write the same data object. A schedule z for transaction system T is said to be *equivalent* to another schedule z^* for T, if for every pair of steps $t_{i,j}$ and $t_{k,m}$ in z and z^*,

$$\forall((t_{i,j}, t_{k,m} \in \cup T) \wedge (O_{t_{i,j}} \equiv O_{t_{k,m}}))[((t_{i,j}, t_{k,m} \in z) \wedge (t_{i,j} > t_{k,m})) \Rightarrow ((t_{i,j}, t_{k,m} \in z^*) \wedge (t_{i,j} > t_{k,m}))]$$

That is, for each of the data objects in the database, O, the orderings of transaction steps induced by z and by z^*, $t_{k,m}(\ldots (t_{i,j}(O))$, are identical.

We now prove an important theorem which states that if z and z^* for T are equivalent and if z is consistent and correct, then z^* is also consistent and correct. We begin our proof with the following lemma.

Lemma 2.1: Let z and z^* be two schedules for transaction system T. Let t be a step of transaction $T_i \in T$. Let the values of the data object accessed by t in z and z^* be O_t and O_t^* respectively. Let the values of the local variable associated with t in z and z^* be L_t and L_t^* respectively. Given the identical initial state X to both z and z^*, we have

$$\forall(T_i \in T)\forall(t \in T_i)((O_t = O_t^*) \wedge (L_t = L_t^*))$$

That is, the values input to and output from any transaction step under z are equal to those under z^*.

Proof: Recall that the syntax of a transaction step is as follows.

$$L_{t_{i,l}} := O_{t_{i,l}}$$

$$O_{t_{i,l}} := f_{t_{i,l}}(L_{t_{i,1}}, \ldots, L_{t_{i,l}})$$

It follows that a transaction step will output identical values under z and z^* if for any transaction step $t \in \cup T$ the values input to t under z and under z^* are equal. Thus, we need to show only that each transaction step $t \in \cup T$ inputs the same values under both schedules z and z^*.

Now consider the first step in schedule z denoted as $t_{z,1}$. Step $t_{z,1}$ must input the initial value of some data object in D denoted as $O_{t_{z,1}}$. By the definition of equivalent schedules and with the same initial state of D, step $t_{z,1}$ must input the same initial value of data object $O_{t_{z,1}}$. Hence, $t_{z,1}$ outputs the same value under both z and z^*. Now consider the second step $t_{z,2}$ in z. The value input to the local variable of $t_{z,2}$ under z is either the initial value of a data object or the value output by step $t_{z,1}$. By the definition of equivalent schedules and by the fact that $t_{z,1}$ outputs the same value under both z and z^*, step $t_{z,2}$ will input the same value under both

z and z^*. Suppose that step $t_{z,h}$ inputs the same value under schedules z and z^*. Following the argument above, step $t_{z,h+1}$ inputs the same value under z and z^*. By induction, this lemma is true. □

Theorem 2.1: Let schedule z be a consistent and correct schedule for transaction system T. If schedule $z^* \equiv z$, then z^* is also consistent and correct.

Proof: To show that z^* is correct, we need to show that for any $T_i \in T$, the execution of T_i under z^* produces correct results. Let the values input to T_i be $O_{i,1}[v_b]$, ..., $O_{i,k_i}[v_b]$. Let the values output by T_i be $O_{i,1}[v_f]$, ..., $O_{i,k_i}[v_f]$. The post-condition of T_i is the specification of the output values of T_i as some functions of the input values: $O_{i,j}[v_f] = f_{i,j}(O_{i,1}[v_b], ..., O_{i,k_i}[v_b])$, $j = 1$ to k_i. It follows from Lemma 2.1 that all the input values to and the output values from T_i under z and z^* are identical. Therefore, the post-condition of T_i must be satisfied under both z and z^*.

To show that z^* is consistent, let the initial state of D be $X \in U$ and the final states of D that result from executing T according to z and z^* be Y and Y^* respectively. It follows the definition of equivalent schedules and Lemma 2.1 that $Y = Y^*$. Thus, Y^* is a consistent state. □

As an example of equivalent schedules, a serializable schedule is defined as a schedule z for which there exists a serial schedule z^* such that $z \equiv z^*$. The consistency and correctness of serial schedules directly follow from the assumption that each transaction is consistent and correct when executing alone. By Theorem 2.1, serializable schedules are also consistent and correct.

2.3 Modular Scheduling Rules

So far our models of database, transactions and schedules are similar to those in the database literature. We now extend our models to address the concepts related to modular scheduling rules. Intuitively, a scheduling rule is a specification of the permissible interleavings of transaction steps in a schedule. For example, serializability theory is a scheduling rule which specifies that in a schedule the steps of a transaction must be interleaved serializably with the steps of other transactions. A scheduling rule specifies the interleavings of transaction steps by partitioning the steps of each transaction into equivalent classes called atomic step segments. The atomic step segments of one transaction must be interleaved serializably with those of other transactions in schedules that satisfy this rule. For example, in a serializable schedule all the steps of a transaction must be interleaved serializably with those of other transactions.

We now define some important properties of scheduling rules. A scheduling rule is said to be consistent and correct if all the schedules that satisfy this rule are consistent and correct. A scheduling rule is said to be modular if it partitions each transaction into atomic step segments independently of other transactions in the system. A modular scheduling rule is said to be application independent if it performs the partition without using the semantic information of transactions. For example, serializability theory is a modular scheduling rule, because it "partitions" each transaction into a single atomic step segment independently of other transactions. It is also an application independent rule, because it performs the "partition" without using any semantic information of transactions.

Given a set of scheduling rules, we are interested in knowing which one provides a higher degree of concurrency. To this end, we model the concurrency of scheduling rules by set containment. If the set of schedules that satisfies scheduling rule R_1 is a subset of that which satisfies R_2, then we say that R_2 is more concurrent than R_1. We consider that a consistent and correct modular scheduling rule is optimal under some conditions if this rule is at least as concurrent as any other consistent and correct modular scheduling rule under the same conditions. For example, serializability theory is optimal under the condition that only transaction syntax information is used for scheduling [Kung 79].

The optimality of scheduling rules is relatively easy to investigate when information used for scheduling is not too rich, such as is the case when only transaction syntax information is used. We will also identify the optimal scheduling rule when the consistency constraints of the database and transaction syntax information are both used for scheduling. However, when the semantic information of transactions is also available for scheduling, the identification of the optimal scheduling rule(s) becomes extremely difficult. In this case, we settle for a less ambitious goal: completeness. This means that we will develop a family of modular scheduling rules for users. This family of rules is complete within the set of consistent and correct modular scheduling rules. This means that given any consistent and correct scheduling rule R^*, a user can always find a rule R in this family such that R is at least as concurrent as R^*.

Having reviewed the concepts related to scheduling rules, we now formally define the concept of scheduling rules in Section 2.3.1, the concept of consistency and correctness of scheduling rules in Section 2.3.2, the concept of modularity and application independence in Section 2.3.3 and finally the concept of optimality and completeness in Section 2.3.4.

2.3.1 Definition of Scheduling Rules

Having discussed the concepts related to modular scheduling rules, we now formalize the concept of a scheduling rule and its relation to schedules.

Let T_m denote the set of all the possible consistent and correct transactions with m steps. Let T denote the set of all the possible consistent and correct transactions, that is, $T = \cup_{m=1}^{\infty} T_m$. Let P_m denote a partition into atomic step segments of a m-step consistent and correct transaction. Let $\mathcal{P}_m$ denote the set of all the possible partitions of a consistent and correct m-step transaction. Let $\mathcal{P}$ be the set of all the possible partitions, that is, $\mathcal{P} = \cup_{m=1}^{\infty} \mathcal{P}_m$.

Definition 2.14-1: A scheduling rule for a transaction system with n transactions, R_n, is a function which takes the transaction system of size n and partitions each of the n transactions.

$$R_n: \prod_{i=1}^{n} T \rightarrow \prod_{i=1}^{n} \mathcal{P}$$

Definition 2.14-2: A scheduling rule R is a function which takes a transaction system of any size and partitions each of the transactions in the system:

$$R: \cup_{n=1}^{\infty} (\prod_{i=1}^{n} T) \rightarrow \cup_{n=1}^{\infty} (\prod_{i=1}^{n} \mathcal{P})$$

such that the restriction of R to $\prod_{i=1}^{n} T$ is R_n. That is, $R|\prod_{i=1}^{n} T = R_n$, n = 1 to ∞

Given a scheduling rule R, we must identify the set of schedules that satisfy R. A schedule z satisfies R if atomic step segments of one transaction are interleaved serializably with those of others in z. This is formalized as follows.

Definition 2.15: Let $T = \{T_1, ..., T_n\}$ be a consistent and correct transaction system, that is, $T \subset T$. Let T_i be a transaction in T. Let $\Xi_R(T_i)$ denote the partition of the steps of T_i by R. Let Γ be the set of all the atomic step segments of T specified by R, that is, $\Gamma = \cup_{T_i \in T} \Xi_R(T_i)$. Let D be the database and Z(T) be the set of all the possible schedules for T. Finally, let σ_i be an atomic step segment of T_i, that is, $\sigma_i \in \Xi_R(T_i)$. A schedule $z \in Z(T)$ is said to be *atomic step segment serial* with respect to R if atomic step segments specified by R and belonging to different transactions do not overlap in z. That is,

$$\forall(T_i, T_j \in T, i \neq j)\ \forall(\sigma_i \in \Xi_R(T_i))\ \forall(\sigma_j \in \Xi_R(T_j))\ \forall(t \in \sigma_i)\ ((t < t_j^1) \vee (t > t_j^m))$$

where t_j^1 and t_j^m are the first and last steps in σ_j respectively.

Definition 2.16: A schedule $z \in Z(T)$ is said to satisfy scheduling rule R, if atomic step segments specified by R and belonging to different transactions are interleaved serializably in z. That is, z satisfies R if and only

$$\exists(z^* \in Z(T))\ ((z \equiv z^*) \wedge (z^* \text{ is atomic step segment serial}))$$

The set of all the schedules in Z(T) that satisfies R is denoted by $Z_R(T)$. That is,

$$Z_R(T) = \{z \in Z(T) |\ z \text{ satisfies } R\}$$

2.3.2 Consistency and Correctness

In the previous section, we have defined the concept of a scheduling rule and its relation to schedules. We now define the key properties of a scheduling rule: consistency and correctness.

Definition 2.17: A scheduling rule R is said to be consistent and correct if and only if all the schedules that satisfy R are consistent and correct.

$$\forall(T \subset \mathbf{T})\forall(z \in Z_R(T))\ (z \text{ is consistent and correct})$$

where T is the set of all the consistent and correct transactions.

In the following, we limit our discussions to consistent and correct scheduling rules.

2.3.3 Modularity and Application Independence

We now define two important properties of a scheduling rule, in addition to the concept of consistency and correctness. These two properties are modularity and application independence. We consider a scheduling rule to be *modular*, if it partitions each transaction in a way that is independent of all the other transactions in the system.

Definition 2.18: A scheduling rule R is said to be *modular* if and only if R partitions each transaction independently. That is,

$R(T_1, ..., T_n) = (R_1^1(T_1), ..., R_1^n(T_n)), n = 1 \text{ to } \infty,$

The scheduling rule for individual transactions, R_1^i, i = 1 to n, will be referred to as the *kernels* of the modular scheduling rule R.

Note that the kernels of a modular scheduling rule need not be identical for different transactions. In other words, a modular scheduling rule can consist of a family of kernels. This allows each of these kernels to take advantage of the semantic information of the given transaction.

We now turn to the concept of an application independent scheduling rule. Thus far we have assumed that each programmer writes and schedules his own transaction. The scheduling is done with full knowledge of the transaction written but without specific knowledge of others' transactions.

To further simplify the scheduling task, we would like to develop scheduling rules that can be implemented by a simple procedure that is independent of the computations carried out by transactions. To this end, an application independent scheduling rule must ignore the specific steps of various transactions. In other words, an application independent scheduling rule views a transaction as a sequence of read and write steps but ignores any detailed aspects of the computation carried out by the transaction. We formalize our discussion on application independent scheduling rules as follows.

Definition 2.19-1: Two transactions T_i and T_j are said to be *syntactically identical* if and only if

1. Transactions T_i and T_j have the same number of steps.
2. If step k of transaction T_i reads (writes) data object O, then step k of transaction T_j reads (writes) the same data object O.

Definition 2.19-2: Two consistent and correct transactions T_i and T_j are said to be *equivalent in syntax* if and only if T_i and T_j both satisfy a given syntactic definition.

Definition 2.20: A modular scheduling rule R is said to be *application independent* if the kernels of R are the same R_1 for all the transactions that are equivalent in syntax. Furthermore, kernel R_1 must produce identical partitions for transactions that are identical in syntax. That is, let $\{T_i, 1 \leq i \leq n\}$ be a set transactions with equivalent syntax. We have

1. $R(T_1, ... T_n) = (R_1(T_1), ... R_1(T_n))$, $n = 1$ to ∞.

2. $R_1(T_i) = R_1(T_j)$, whenever transactions T_i and T_j are syntactically identical.

Conceptually, an application independent rule is a procedure (function) that can be applied to all the transactions defined by a given syntax. Furthermore, this procedure views a transaction simply as a sequence of read and write steps. For example, the serializable scheduling rule is a modular rule with the same scheduling rule kernel for all the transactions, which takes the entire set of steps of a transaction as a single atomic step segment. This procedure certainly produces identical partitions for syntactically identical transactions. Thus, this rule is not only modular but also application independent.

2.3.4 Optimality and Completeness

We have defined four important properties of a scheduling rule, namely consistency, correctness, modularity and application independence. In this section, we come to address issues related to the degree of concurrency provided by scheduling rules. We address these issues using the concepts of optimality and completeness. Before formally defining these two concepts, we first comment on the implications of using a richer set of primitive steps in addition to the "read" and "write" steps used in this work.

Korth [Korth 83] has shown that for serializable schedules the concurrency can be improved if the set of primitive steps is expanded to include other commutative ones. The idea is that if a set of steps is commutative, then there is no need to control their relative order. The reason is as follows. We know that if in schedule z the ordering of the steps on each of the data objects in the database, $t_{k,m}(... (t_{i,j}(O))$, is consistent with the ordering of steps of some serial schedule z^*, then schedule z is serializable. Suppose that in schedule z some of the steps are "out of order" but they are commutative. Without affecting the results of computations we can always rearrange the ordering of these steps in such way that the ordering of these steps becomes consistent with that in a serial schedule. Therefore, scheduling z is serializable. In the following, we limit our discussion to transactions using only primitive steps: "read" and "write". The use of commutative steps to improve the concurrency of (generalized) setwise serializable schedules can be done in a manner similar to that done by Korth for serializable schedules.

We begin our investigation by first defining a way to compare the degree of concurrency offered by different scheduling rules.

Definition 2.21: Scheduling rule R^1 is said to be *at least as concurrent as* R^2, denoted by $R^1 \geq R^2$, if and only if,

$$\forall (T \subset \mathcal{T})(Z_{R^2}(T) \subseteq Z_{R^1}(T))$$

That is, the concurrency of schedules is partially ordered by set containment. The relative concurrency of two scheduling rules can be incomparable. We now define the concept of an optimal application independent scheduling rule.

Definition 2.22: Let Λ_A be the set of all the application independent scheduling rules. An application independent scheduling rule $R^* \in \Lambda_A$ is said to be optimal if R^* is at least as concurrent as any rule in Λ_A. That is,

$$\forall (R \in \Lambda_A)(R^* \geq R)$$

We now introduce the concept of *completeness*. A family of modular scheduling rules is said to form a complete class within the set of modular scheduling rules if and only if given any modular scheduling rule R we can always find a rule R^* in this family of rules such that R^* is at least as concurrent as R.

Definition 2.23: Let Λ_M be the set of all the consistent and correct modular scheduling rules. A set of consistent and correct modular scheduling rules $\mathfrak{R}$ is said to form a *complete class* within Λ_M, if and only if

$$\forall (R \in \Lambda_M)[\exists (R^* \in \mathfrak{R})(R^* \geq R)]$$

2.4 Summary

In this chapter, we have formally defined our model of modular scheduling rules. In Sections 2.1 and 2.2, we provided a formal description of the standard concepts in the study of transaction systems such as database, transactions and schedules. In Section 2.3, we defined the concepts related to modular scheduling rules and their important properties. Having formally defined these basic concepts, we now move on to the investigation of the structure of the database that can be used for the purpose of concurrency control in the light of the knowledge of database consistency constraints.

3. Atomic Data Sets

In Chapter 2, we developed a formal model of modular scheduling rules. This model provides us with the theoretical basis for the development of new concurrency control rules. We now develop a major concept called *atomic data sets*, which is important to the development of modular concurrency control and failure recovery rules. Intuitively, an atomic data set is simply a set of data objects with a set of associated consistency constraints that can be satisfied independently of other atomic data sets. For example, in a distributed computer system a job queue at a computer waiting for execution is an atomic data set, the local directory to a computer's files is an atomic data set, and each user's mailbox is also an atomic data set. This is because the consistency of each of these entities is defined by a set of consistency constraints that can be satisfied independently of others.

It is important to note that inter-related data objects can be in different atomic data sets, as long as the consistency of an atomic data set can be satisfied independently of other atomic data sets. For example, we can move a job waiting on the job queue of one computer to the job queue of another computer in order to balance the workloads. That is, these job queues are related by some load balancing transactions. However, at each computer a load balancing transaction must observe the consistency constraints of the local job queue such as the maximum size and capability constraints. The capability constraints associated with a job queue are a specification of the types of jobs that are allowed to execute on a computer.

Given a set of consistency constraints of the database (shared system data objects), we show in Theorem 3.2 of this chapter that there always exists a unique maximal consistency preserving partition of the database. That is, there is always a unique way to partition the database into the largest possible number of atomic data sets when the consistency constraints are given. In Chapter 4, we will show that if each transaction is consistent and correct when executed alone, then each transaction will also be executed consistently and correctly when it is executed *setwise serializably*. That is, when the database is partitioned into atomic data sets, we only need to enforce serializability of concurrency control on a set by set basis. To improve system concurrency, we would like to have many small atomic data sets instead of a few large ones in the design of the database. Theorem 3.2 of this chapter indicates that the number of atomic data sets we can have is determined by the consistency constraints. Therefore, there is a motivation to design "weaker" consistency constraints in a distributed computer system in order to explore the potential concurrency provided by distributed hardwares. In Section 7.2, we use an example to illustrate why distributed computer systems call

for an approach to specify consistency constraints that is different from the approach which we have got accustomed to in the design of centralized computer systems. Having introduced the relationship between concurrency control and the concept of atomic data sets, we now formalize this concept in this chapter.

3.1 Consistency Preserving Partition of The Database

In this section, we first define the concepts of atomic data sets and a consistency preserving partition of the database. Next, we show that the conjunction of the consistency constraints of the atomic data sets is equivalent to that of the database. Finally, we prove that there is always a unique maximal consistency preserving partition with respect to a given set of consistency constraints.

Definition 3.1-1: Let $I = \{1, 2, ... n\}$ be the index set of database D. The index $i \in I$ specifies the data object $O_i \in D$. Let $\pi_s(Y)$ denote the projection of an n-tuple $Y \in \Omega$ using the set of indices $S \subset I$. That is, $\pi_s(Y)$ denotes the tuple whose elements are the values of the data objects indexed by S. Let $P = \{S_1, ... , S_k\}$ denote a partition of I. Let V_i be the set whose elements are the projections of all the consistent states, $X \in U$, onto an arbitrary index set S_i, that is, $V_i = \bigcup_{X \in U} \{\pi_{S_i}(X)\}$. A partition of the index set I, $P = \{S_1, S_2, ..., S_k\}$, is said to be consistency preserving (CP) if and only if,

$$\forall (Y \in \Omega)\{ [\pi_{S_i}(Y) \in V_i, i = 1 \text{ to } k] \rightarrow [Y \in U] \}.$$

Definition 3.1-2: An atomic data set $\mathcal{A}_i$ for a CP partition P is the set of data objects specified by $S_i \in P$. The associated partition of data objects in D, Q, is called a consistency preserving partition of D.

The definition of a CP partition states that a CP partition has the property that any choice of the consistent states of the atomic data sets leads to a consistent state of the database. We now introduce the concept of consistency constraints of an atomic data set. Next, we show that the consistency constraints of the database can be decomposed into sets of ADS consistency constraints.

Definition 3.2: Let P be a CP partition of I and let $S_i \subset I$ be the set of indices which specify the data objects in the atomic data set $\mathcal{A}_i \subset D$. Let the set of all the consistent states of an ADS $\mathcal{A}_i$ be $U_i = \bigcup_{X \in U}\{\pi_{S_i}(X)\}$. The set of consistency constraints C_i whose truth set is the consistent states of $\mathcal{A}_i$ is called the ADS consistency constraints of $\mathcal{A}_i$. That is,

Proof: If $S = I$, then $H_S(X_1, X_2) = X_2$, and the result follows.

Let $P = \{S, \sigma_1, \ldots, \sigma_k\}$, $k \geq 1$ be a CP partition.

Define $W_0 = X_2$, $W_i = X_1$, $i = 1$ to k; so that $W_i \in U$, $i = 0$ to k.

$\pi_S(W_0) = \pi_S(H_S(X_1, X_2))$

$\pi_{\sigma_i}(W_i) = \pi_{\sigma_i}(H_S(X_1, X_2))$, $i = 1$ to k.

Given that P is CP, $H_S(X_1, X_2)$ is therefore in U by the definition of a CP partition. Thus H_S maps pairs of consistent state into a consistent state. □

When we have two or more distinct CP partitions of the same index set I, sets from distinct partitions could intersect. Lemma 3.2-2 generalizes Lemma 3.2-1 by allowing the intersections to be used for the specification of swapping. For example, let $P_2 = \{ \{1\}, \{2, 3\} \}$ be a second CP partition. The intersection of $\{1, 2\} \in P_1$ and $\{2, 3\} \in P_2$ is $\{2\}$. Lemma 3.2-2 states that the two states resulting from swapping the projections of A and B specified by $\{2\}$ are also consistent. That is, (a_1, b_2, a_3) and (b_1, a_2, b_3) are consistent. We show the consistency of (a_1, b_2, a_3) as follows. First, we use Lemma 3.2-1 to swap the projections of A and B specified by $\{1\}$ of P_2. One of the two resulting consistent states is $E = (a_1, b_2, b_3)$. Next, swapping the projections of A and E specified by $\{3\}$ of P_2, we find (a_1, b_2, a_3) to be a consistent state also. We now give a general proof of Lemma 3.2-2.

Lemma 3.2-2: Suppose that $S \in P_1$ and $\sigma \in P_2$, where P_1 and P_2 are CP. Then $H_{S \cap \sigma}: U \times U \rightarrow U$.

Proof: If $S \in P_1$, then there exists a CP partition P such that $S^c \in P$. Since $X_1, X_2 \in U$, it follows that $H_{S^c}(X_1, X_2) \in U$ by Lemma 3.2-1. Therefore, $H_\sigma(X_1, H_{S^c}(X_2, X_1)) \in U$ as well. The Lemma follows, since $H_\sigma(X_1, H_{S^c}(X_2, X_1)) = H_{S \cap \sigma}(X_1, X_2)$. □

Lemma 3.2-3 demonstrates that the least common refinement of any two CP partitions is also a CP partition. For example, the least common refinement of P_1 and P_2, $\{\{1\}, \{2\}, \{3\}\}$, is also CP. In this case, given a set of consistent states such as (a_1, a_2, a_3), (b_1, b_2, b_3) and (c_1, c_2, c_3), we must prove that $\{a_1, b_2, c_3\}$ is also consistent. First, we apply the intersection of $\{1\}$ and $\{1, 2\}$ to "A, B" and "A, C" respectively. States $(a_1,$

$$U_i = \{ \pi_{S_i}(Y) \mid C_i(\pi_{S_i}(Y)) = 1 \}$$

Theorem 3.1: The conjunction of all the ADS constraints C_i, i = 1 to k, is equivalent to consistency constraints C of D. That is,

$$C = C_1 \wedge C_2 \wedge \ldots C_k.$$

Proof: Let U^* be the truth set of the conjunction of all the ADS constraints. We have,

$$U^* = \{Y \mid C_i(\pi_{S_i}(Y)), i = 1 \text{ to } k\}$$

$$= \{Y \mid \pi_{S_i}(Y) \in U_i, i = 1 \text{ to } k\} = U.$$

Hence, $C = C_1 \wedge C_2 \wedge \ldots C_k$. □

We now prove the theorem that there always exists a unique maximal CP partition. First, we note that CP partitions exist, since the trivial partition { I } is CP. Furthermore, the CP partitions are partially ordered by refinement. That is, for any pair of CP partitions P_1 and P_2, partition P_1 is refined by P_2 if and only if $\forall(S_j \in P_2)\ \exists(S_i \in P_1)(S_j \subset S_i)$. A maximal CP partition is one which is refined by no other CP partition. In the following, we will prove that there exists a unique maximal CP partition P. Since the proof is relatively complex, we would like first to illustrate the ideas of the proof.

The proof is based on three lemmas. The idea of Lemma 3.2-1 is illustrated by the following example. Let $P_1 = \{ \{1, 2\}, \{3\} \}$ be a CP partition of the index set $I = \{1, 2, 3\}$. Suppose that $A = (a_1, a_2, a_3)$ and $B = (b_1, b_2, b_3)$ are two consistent states. Let S be a partition set, either {1, 2} or {3}. Lemma 3.2-1 states that the two new states which result from swapping the projections of A and B specified by S are also consistent. That is, (a_1, a_2, b_3) and (b_1, b_2, a_3) are consistent.

We now define a mapping $H_S: \Omega \times \Omega \rightarrow \Omega$ as follows, where $S \subset I$. Given that $X_1, X_2 \in \Omega$, $H_S(X_1, X_2) = Y$, where Y satisfies $\pi_S(Y) = \pi_S(X_2)$ and $\pi_{S^c}(Y) = \pi_{S^c}(X_1)$, where $S^c = I - S$. Thus $H_S(X_1, X_2)$ replaces the projections of X_1 specified by S with the projections of X_2 specified by S.

Lemma 3.2-1: Suppose that $X_1, X_2 \in U$. If S is an element of any CP partition of I, then $H_S: U \times U \rightarrow U$.

software modules: modules that encapsulate distributed data objects. For example, one can generalize the monitor (or abstract data type) approach developed for centralized systems as follows. We can define a set of primitive procedures for each of the data objects in an atomic data set to facilitate the manipulation of these data objects. When the scheduling rules grant one the privilege, one is entitled to use those pre-defined procedures as building blocks of one's own transaction. In the next chapter, we show how the concept of atomic data sets can be used for concurrency control.

b_2, b_3) and (a_1, c_2, c_3) are two of the four new consistent states. Next, we apply the intersection of {2, 3} and {3} to these two new states. One of the two resulting consistent states is $\{a_1, b_2, c_3\}$. We now give a general proof of Lemma 3.2-3.

Lemma 3.2-3 if P_1 and P_2 are CP, then their least common refinement is also CP.

Proof: Let $P_1 = \{S_1, \ldots, S_m\}$ and $P_2 = \{\sigma_1, \ldots, \sigma_n\}$. Their least common refinement is $P_1 \cap P_2 = \{C_1, \ldots, C_L\}$, where $C_i = S_j \cap \sigma_k$ for some j, k, i = 1 to L.

Let $X_i \in U$, i = 1 to L and $Y \in \Omega$ be given such that $\pi_{C_i}(X_i) = \pi_{C_i}(Y)$, i = 1 to L. We must prove $Y \in U$ to conclude that the $P_1 \cap P_2$ is CP.

To this end, we define a sequence $\{X^*_i, i = 1 \text{ to } L\}$ as follows: $X_1^* = X_1$, $X_j^* = H_{C_j}(X^*_{j-1}, X_j)$, j = 2 to L. Noting that $C_j = S_i \cap \sigma_k$, Lemma 3.2-2 indicates that $X_j^* \in U$, j = 1 to L. It follows that $X^*_L = Y \in U$. □

Theorem 3.2: There exists a unique maximal CP partition.

Proof: Suppose that there exists more than one maximal CP partition. The least common refinement of distinct maximal CP partitions is CP by Lemma 3.2-3, thus contradicting the maximality assumption. □

Corollary 3.2: There exists a unique maximum CP partition Q of D.

In the next chapter, we show how consistency preserving partitions can be used to schedule transactions in a non-serializable fashion. Although any CP partition can be used, Theorem 3.2 indicates that there is a CP partition that is "most refined" with respect to a given set of consistency constraints. This partition will allow the maximal concurrency in concurrency control.

3.2 Summary

In this chapter, we have defined the concepts of atomic data sets and of a *consistency preserving* partition of the database. We have also shown that there exists a unique maximal consistency preserving partition of the database with respect to the given set of consistency constraints.

From an application point of view, atomic data sets can also be used as a basis for constructing distributed

4. The Setwise Serializable Scheduling Rule

An objective of this work is to deliver provably consistent, correct, modular and optimal scheduling rules. Having developed a model of modular concurrency control rules and the concept of atomic data sets, we are now in a position to introduce such a rule.

Conceptually, the major result of this chapter can be stated as follows. When the database is partitioned into consistency preserving atomic data sets, we only need to enforce the serializability of concurrency control on a set by set basis rather than globally. As long as each transaction is consistent and correct when executing alone, it will be executed consistently and correctly provided that it is executed setwise serializably. Furthermore, this is also the best we can do in the absence of transaction semantic information.

This conceptual understanding is formally expressed as the optimal application independent scheduling rule: *the setwise serializable scheduling rule*. Given a CP partitioned database and a transaction, the setwise serializable scheduling rule partitions the steps of the given transaction into atomic step segments in the form of transaction ADS segments. A transaction ADS segment in a transaction is simply all the steps in this transaction accessing the same atomic data set. We prove that the setwise serializable scheduling rule is consistent, correct and modular. In addition, we prove that this rule is the optimal application independent scheduling rule. In other words, this is the rule that provides the highest degree of concurrency and does not use any transaction semantic information. The term "application independent" refers to the fact that this rule can be applied to transactions by an algorithm which does not know the semantics of transactions.

This chapter is organized as follows. In Section 4.1, we define the concepts of setwise serializable schedules and the setwise serializable scheduling rule. In Section 4.2, we prove that this rule is consistent and correct in the context of single level transactions. In Section 4.3, we generalize these results to nested transactions. In Section 4.4, we prove that this rule is application independent and optimal. Finally, we summarize this chapter in Section 4.5.

4.1 Definitions

In this section, we first define the concept of a transaction ADS segment. We then define the scheduling rule called the *setwise serializable* scheduling rule. Finally, we define the set of schedules that satisfy this rule.

Definition 4.1: A transaction ADS segment is the sequence of steps in a transaction that read or write data objects in the same ADS. Let $\Psi(i, \mathcal{A})$ denote the transaction ADS segment of transaction T_i accessing ADS $\mathcal{A}$. Let $t_{i,j} > t_{k,m}$ denote that step $t_{i,j}$ is executed after step $t_{k,m}$. We have

1. $\Psi(i, \mathcal{A}) = \{t \mid (t \in T_i) \wedge (t \text{ reads or writes a data object in ADS } \mathcal{A})\}$

2. $\forall ((t_{i,j}, t_{i,k} \in \Psi(i, \mathcal{A})) \wedge (t_{i,j} > t_{i,k}))((t_{i,j}, t_{i,k} \in T_i) \wedge (t_{i,j} > t_{i,k}))$

Definition 4.2: The scheduling rule that partitions each of the transactions in a transaction system T into transaction ADS segments is called the *setwise serializable* scheduling rule.

A setwise serial schedule z for a transaction system T is a schedule in which transaction ADS segments accessing the same ADS do not overlap. This concept is formalized as follows.

Definition 4.3: Let $t_i^{A.1}$ and $t_i^{A.m}$ denote the first step and the last step of transaction ADS segment $\Psi(i, \mathcal{A})$ respectively. Let Q be a consistency preserving partition of D. A schedule z for transaction system T is said to be setwise serial if and only if under z,

$$\forall (\mathcal{A} \in Q) \forall (T_i \in T) \forall (t^A \in z \wedge t^A \notin T_i)((t_i^{A.1} > t^A) \vee (t^A > t_i^{A.m}))$$

where t^A represents any step accessing ADS $\mathcal{A}$ in the transaction system T.

Having defined the concept of setwise serial schedules, we now define a setwise serializable schedule as one which is equivalent to a setwise serial schedule.

Definition 4.4: A schedule z for transaction system T is said to be setwise serializable if there exists a setwise serial schedule z^* for T such that $z \equiv z^*$.

4.2 Consistency and Correctness

We now prove that setwise serializable schedules are consistent and correct. The proof is organized into three lemmas. Let T_i be a consistent and correct transaction. In Lemma 4.1-1, we prove that T_i preserves the consistency of each of the accessed atomic data sets and produces correct results when executing alone. In Lemma 4.1-2, we further prove that at the end of executing a transaction ADS segment $\Psi(i, \mathcal{A})$ of T_i, the consistency of $\mathcal{A}$ has been already preserved. In addition, the output values of data objects in $\mathcal{A}$ are correct at

the end of $\Psi(i, \mathcal{A})$. We need not wait for the end of T_i to know these results. In Lemma 4.1-3, we relax the executing alone condition. We show that the results of Lemma 4.1-2 are still valid for any ADS $\mathcal{A}$, as long as $\mathcal{A}$ is consistent at the beginning of transaction segment $\Psi(i, \mathcal{A})$.

Definition 4.5: An ADS $\mathcal{A}_j$ is said to be accessed by a transaction, if this transaction reads or writes one or more data objects in $\mathcal{A}_j$.

Lemma 4.1-1: Let $Q = \{\mathcal{A}_1, ..., \mathcal{A}_k\}$ be a given CP partition of D. Let T_i be a consistent and correct transaction. If T_i executes alone and if the states of the atomic data sets accessed by T_i are initially consistent, then at the end of T_i the state of each of the accessed atomic data sets is consistent, and the values output by T_i satisfy the post-condition of T_i.

Proof: let $\mathcal{A}_{i,j}$, j = 1 to k_i, be the atomic data sets accessed by transaction T_i. Let $Y \in \Omega$ be a state of D such that $C_{i,j}(\pi_{S_{i,j}}(Y)) = 1$, j = 1 to k_i; where $C_{i,j}$ represents the ADS consistency constraints of $\mathcal{A}_{i,j}$, and $S_{i,j}$ represents the index set of $\mathcal{A}_{i,j}$. Now let X be a consistent state of the database such that $\pi_{S_{i,j}}(X) = \pi_{S_{i,j}}(Y)$, j = 1 to k_i. Next, we let T_i execute alone with the database initially in state X. We now prove that with either X or Y as initial state, the executions of T_i are equivalent.

By assumptions A1 to A3 in Section 2.2.1, with X as initial state, T_i produces correct results and preserves the consistency of the database. It follows that T_i preserves the consistency of all the atomic data sets. To complete the proof, we must show that the values of the local variables and the values of the global variables of T_i are identical in both schedules.

Let the values of the local variable and the data object in the execution with initial state X be $L^{\bullet}_{t_{i,l}}$ and $O^{\bullet}_{t_{i,l}}$. Let the values of the local variable and the data object with initial state Y be $L_{t_{i,l}}$ and $O_{t_{i,l}}$. We must prove that $L_{t_{i,l}} = L^{\bullet}_{t_{i,l}}$ and $O_{t_{i,l}} = O^{\bullet}_{t_{i,l}}$ at each step of the transaction. Recall that the syntax of a transaction step is as follows,

$$L_{t_{i,l}} := O_{t_{i,l}}$$

$$O_{t_{i,l}} := f_{t_{i,l}}(L_{t_{i,1}}, ..., L_{t_{i,l}})$$

Since the initial states of all accessed ADS are equal with either X or Y as the initial state, it follows that the

initial values of the accessed data objects are equal. Hence, at the first step of T_i, $L_{t_{i,1}} = L^*_{t_{i,1}}$. In addition, $O_{t_{i,1}} = f_{t_{i,1}}(L_{t_{i,1}}) = f_{t_{i,1}}(L^*_{t_{i,1}}) = O^*_{t_{i,1}}$. Next, $L_{t_{i,2}} = L^*_{t_{i,2}}$, because the second step either reads the initial value of a data object or the value of the data object output by step 1. Similarly, $O_{t_{i,2}} = O^*_{t_{i,2}}$.

Now suppose that these local variable and data object value pairs are equal from steps 1 to r. That is, $L_{t_{i,h}} = L^*_{t_{i,h}}$ and $O_{t_{i,h}} = O^*_{t_{i,h}}$, h = 1 to r. We show that $L_{t_{i,r+1}} = L^*_{t_{i,r+1}}$. This follows because step r+1 either reads the initial value of a data object or a data object which has been output by some step between 1 to r. It follows that $O_{t_{i,r+1}} = O^*_{t_{i,r+1}}$. By induction, the final values of accessed data objects with either X or Y as initial state are equal. Since the values of data objects in $\mathcal{A}_{i,j}$, j = 1 to k_i, not accessed by T_i remain unchanged, they must be equal at the end of the transaction with either X or Y as the initial state. That is, $\pi_{S_{i,j}}(W_x) = \pi_{S_{i,j}}(W_y)$, j = 1 to k_i, where W_x and W_y are the states of D at the end of executing T_i with X and Y as initial states respectively. Since the execution using X as the initial state is assumed to preserve the consistency of each of the accessed atomic data sets, the execution using Y as the initial state must also preserve the consistency of each of the accessed atomic data sets. Since the execution with X as the initial state produces correct results, the execution using Y as the initial state must also produce correct results. □

There are two implications from Lemma 4.1-1. First, after the decomposition of the database, a programmer is only required to know and maintain the consistency constraints of the atomic data sets accessed by his transaction. Without a CP partition, everyone must, in principle, know all the database consistency constraints in order to verify that one's transaction will not violate any of them. Second, this lemma implies that a transaction can still function properly as long as the accessed atomic data sets are consistent, even if the rest of the atomic data sets are inconsistent. This is useful in fault isolation, although this topic is outside the scope of this work. We now turn to the second lemma.

Lemma 4.1-2: Let the atomic data sets accessed by transaction T_i be $\mathcal{A}_{i,j}$, j = 1 to k_i. If $\mathcal{A}_{i,j}$, j = 1 to k_i, are initially consistent and if T_i executes alone, then at the end of transaction ADS segment $\Psi(i, \mathcal{A}_{i,j})$ the consistency of $\mathcal{A}_{i,j}$ is preserved. Furthermore, the values of data objects in $\mathcal{A}_{i,j}$ output by $\Psi(i, \mathcal{A}_{i,j})$ are correct at the end of $\Psi(i, \mathcal{A}_{i,j})$.

Proof: At the end of the transaction ADS segment $\Psi(i, \mathcal{A}_{i,j})$, j = 1 to k_i, the data objects in $\mathcal{A}_{i,j}$, j = 1 to k_i are neither read or written again. It follows that the values of the data objects in $\mathcal{A}_{i,j}$, j = 1 to k_i, are the same as at the end of the transaction. By Lemma 4.1-1, at the end of the transaction, the consistency of each of the

atomic data sets is preserved, and the values of the data objects output by T_i are correct, it follows that at the end of each of the transaction ADS segments the state of the accessed atomic data set is consistent and the values of the data objects output by the segment are correct. □

Lemma 4.1-3: In a setwise serial schedule, if at the beginning of a transaction ADS segment, $\Psi(i, \mathcal{A}_{i,j})$, $= 1$ to k_i, ADS $\mathcal{A}_{i,j}$ is initially consistent, then $\mathcal{A}_{i,j}$ is consistent at the end of $\Psi(i, \mathcal{A}_{i,j})$, and the values of each of the data objects in $\mathcal{A}_{i,j}$ output by T_i are correct.

Proof: Let the atomic data sets accessed by T_i be $\mathcal{A}_{i,j}$, $j = 1$ to k_i. Now let T_i execute alone in a serial schedule z^* with the initial states of $\mathcal{A}_{i,j}$, $j = 1$ to k_i, being identical to the initial states of $\mathcal{A}_{i,j}$, $j = 1$ to k_i, in the setwise serial schedule z.

Let the values of the local variables and data objects in the serial schedule z^* be $L^*_{t_{i,l}}$ and $O^*_{t_{i,l}}$; and those in setwise serial schedule z be $L_{t_{i,l}}$ and $O_{t_{i,l}}$. We now prove that the executions of T_i under z and z^* are equivalent. Recall that the syntax of a transaction step is given by

$$L_{t_{i,l}} := O_{t_{i,l}}$$

$$O_{t_{i,l}} := f_{t_{i,l}}(L_{t_{i,1}}, \ldots, L_{t_{i,l}})$$

Since the initial states of ADS $\mathcal{A}_{i,j}$, $j = 1$ to k_i, are equal in both schedules, the initial values of all the data objects in $\mathcal{A}_{i,j}$, $j = 1$ to k_i, are equal. Therefore, the first steps in both schedules input the same value. That is, $L_{t_{i,1}} = L^*_{t_{i,1}}$. In addition, $O_{t_{i,1}} = f_{t_{i,1}}(L_{t_{i,1}}) = f_{t_{i,1}}(L^*_{t_{i,1}}) = O^*_{t_{i,1}}$. Next, $L_{t_{i,2}} = L^*_{t_{i,2}}$, because step two either reads the initial value of a data object or the value of the data object output by step 1. Similarly, $O_{t_{i,2}} = O^*_{t_{i,2}}$.

Now suppose that these local variable and data object value pairs are equal from steps 1 to r. That is, $L_{t_{i,h}} = L^*_{t_{i,h}}$ and $O_{t_{i,h}} = O^*_{t_{i,h}}$, $h = 1$ to r. We show that $L_{t_{i,r+1}} = L^*_{t_{i,r+1}}$. This follows because step $r+1$ either reads the initial value of a data object or a data object which has been output by some steps between 1 to r. It follows that $O_{t_{i,r+1}} = O^*_{t_{i,r+1}}$. Therefore, the final values of accessed data objects in both schedules are equal at the end of each transaction ADS segment. In addition, data objects in $\mathcal{A}_{i,j}$, $j = 1$ to k_i, not accessed by $T_{i,j}$ remain unchanged and therefore equal at the end of each transaction ADS segment for both schedules. It follows from Lemma 4.1-2 that at the end of $\Psi(i, \mathcal{A}_{i,j})$, the $\mathcal{A}_{i,j}$, $j = 1$ to k_i, are consistent, and the value of each of the data objects in $\mathcal{A}_{i,j}$, $j = 1$ to k_i, output by T_i is correct. □

Theorem 4.1: A setwise serial schedule is consistent and correct.

Proof: Let $Q = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ be a CP partition of D. Let the initial states of each of the ADS's be $Z_{A_j}[0]$, j = 1 to k. These initial states are assumed to be consistent.

Since a schedule is a totally ordered set of steps from all the transactions, each of which terminates, there must exist a transaction ADS segment $\Psi(i, \mathcal{A}_j)$ which first finishes its computation. Let the associated ADS state be $Z_{A_j}[1]$. Since there are no interleavings among transaction ADS segments accessing the same ADS in a setwise serial schedule, $Z_{A_j}[1]$ must be output by a transaction which has used only the initial states that were assumed to be consistent. By Lemma 4.1-3, $Z_{A_j}[1]$ is consistent, and the values of data objects in $\mathcal{A}_j$ output by $\Psi(i, \mathcal{A}_j)$ are correct. Consider now the output of the second transaction ADS segment produced by the schedule. Since it can use only $Z_{A_j}[1]$ or $Z_{A_m}[0]$, m = 1 to k and $m \neq j$, at the end of this second transaction ADS segment, the accessed atomic data set is in a consistent state and the output values are correct by Lemma 4.1-3. Now assume that the first n transaction ADS segments produce consistent and correct results. The $n+1^{st}$ must also by the same argument. By induction, the ADS state produced by each of the transaction ADS segments is consistent, and the values of the data objects output in each ADS at the end of the transaction ADS segment satisfy the post-condition. It follows that a setwise serial schedule is consistent and correct. □

Corollary 4.1-1: Setwise serializable schedules are consistent and correct.

Proof: It directly follows from Definition 4.4 and Theorem 2.1. □

Corollary 4.1-2: The setwise serializable scheduling rule is consistent and correct.

Proof: It directly follows from Corollary 4.1-1. □

Throughout this section, the choice of atomic data sets has been arbitrary, because Theorem 4.1 applies to any CP partition whether maximal or not. If the CP partition consists of a single ADS, then setwise serializable schedules reduce to serializable schedules.

4.3 Nested Transactions

In the early work on serializability theory, a transaction was modelled as a sequence of steps. However, it is natural to write transactions in a nested form, in which sub-transactions can be executed in parallel and invoked by higher level ones. Recently, serializable nested transactions have been studied by Moss, Lynch and Beeri et al. [Moss 81, Lynch 83a, Beeri 83]. We have proved the consistency and correctness of setwise serializable schedules in the context of single level transactions. In this section, we now generalize this result to nested transactions.

4.3.1 Syntax

We can visualize a nested transaction as being organized in the form of a tree whose nodes are sub-transactions and whose leaves are steps. The execution of the transaction is defined by the partial order of the tree. The syntax of a nested transaction is formally defined as follows.

Definition 4.6: A nested transaction, T_i^N, is a partially ordered set of steps with the following syntax:

<NestedTransaction> ::= BeginTransaction <NestedTransactionBody> EndTransaction.

<NestedTransactionBody> ::= BeginSerial <SubTransactionList> EndSerial |
BeginParallel <SubTransactionList> EndParallel

<SubTransactionList> ::= <SubTransaction> | <SubTransactionList><SubTransactionList>

<SubTransaction> ::= Step | <SubTransaction>;<SubTransaction> | <NestedTransactionBody>

A step is modelled as an indivisible execution of the following two instructions.

$$L_{t_{i,j,k}} := O_{t_{i,j,k}}$$

$$O_{t_{i,j,k}} := f_{t_{i,j,k}}(\{L_{t_{i,j,k}}\} \cup \{L_{t_{i,m,n}} \mid t_{i,m,n} < t_{i,j,k}\})$$

where $t_{i,j,k}$ is the k^{th} step at level j of transaction T_i^N, $L_{t_{i,j,k}}$ is the local variable used by step $t_{i,j,k}$, $O_{t_{i,j,k}}$ is the data object accessed by step $t_{i,j,k}$, and $f_{t_{i,j,k}}$ represents the computation performed by step $t_{i,j,k}$. Note that step

$t_{i,j,k}$ can use not only its own local variable but also the local variables associated with steps preceding it. In this model, a "read" step is interpreted as one which writes the original value back into the data object. That is, $f_{t_{i,j,k}}$ is the identity function.

4.3.2 Consistency and Correctness

In this section, we address the consistency and correctness of the setwise serializable scheduling rule when it applies to nested transactions. In a nested transaction, the ordering between some steps is unspecified due the partial ordering. This implies that the results produced by any total ordering that is consistent with the partial ordering in the transaction must be equally valid. Otherwise, one should specify the order. Formally, the issues caused by partial ordering are addressed via the concept of the linearization of a nested transaction.

Definition 4.7: A single level transaction T_i^S is the linearization of the nested transaction T_i^N, if and only if T_i^S has the same steps as T_i^N and if the total ordering of steps in T_i^S is consistent with the partial ordering of steps in T_i^N. That is,

$$[\forall(t)((t \in T_i^S) \Leftrightarrow (t \in T_i^N))] \wedge [\forall((t_k, t_m \in T_i^N) \wedge (t_m > t_k))((t_k, t_m \in T_i^S) \wedge (t_m > t_k))]$$

Definition 4.8: A nested transaction is said to be consistent and correct, if and only if each of the linearizations of the nested transaction, when executed alone satisfies our three assumptions about a transaction: it terminates (A1), preserves the consistency of the database (A2), and produces correct results(A3). In the following, we limit our investigation only to consistent and correct nested transactions.

We now define the concept of a nested transaction system and its schedules.

Definition 4.9: A nested transaction system $T^N = \{T_1^N, ..., T_m^N\}$ is a finite set of nested transactions operating upon the shared database D.

Definition 4.10: Let T_i^S be the set of all the linearizations of nested transaction $T_i^N \in T^N$. Let T^S be a linearized transaction system for T^N. That is, $T^S = \{T_1^S, ..., T_m^S\}$ is a transaction system in which $T_i^S \in T_i^S$, i = 1 to m. Let $\mathbf{T}^S$ be the set of all the linearized transaction systems for T^N. A schedule z for a nested transaction system T^N is a a schedule of a linearized transaction system $T^S \in \mathbf{T}^S$.

Theorem 4.2: Setwise serial schedules of a nested transaction system are consistent and correct.

Proof: By definition, when each of the linearizations of a nested transaction executes alone, it terminates, preserves the consistency of the database and produces correct results. It follows from Theorem 4.1 that a setwise serial schedule for a linearized nested transaction system is consistent and correct. This is true for all the linearizations of the given nested transaction system. It follows that setwise serial schedules for nested transaction systems are consistent and correct. □

Corollary 4.2-1: Setwise serializable schedules for nested transactions are consistent and correct.

Corollary 4.2-2: The setwise serializable scheduling rule is consistent and correct with respect to nested transactions.

4.4 Application Independence and Optimality

In the previous section, we proved the consistency and correctness of the setwise serializable scheduling rule. We now prove that this rule is also the optimal application independent scheduling rule.

Theorem 4.3: The setwise serializable scheduling rule is modular.

Proof: It directly follows from Definitions 2.18 and 4.2. □

Theorem 4.4: The setwise serializable scheduling rule is application independent.

Proof: The kernel of the setwise serializable scheduling rule is the function that partitions the steps of any given transaction into transaction ADS segments. Transactions which are syntactically identical are partitioned identically. It follows from Definition 2.20 that the setwise serializable scheduling rule is application independent. □

We now prove that the setwise serializable scheduling rule is optimal in the set of application independent scheduling rules. To illustrate the basic idea of the proof, let us consider an example first. Suppose that we have an atomic data set, $\mathcal{A}$, with two data objects A and B. The consistent states of this ADS are (0, 1) and (1, 0). Suppose that transaction T_1 and T_2 assigns (0, 1) and (1, 0) to $\mathcal{A}$ respectively. Let Transaction T_3 be a transaction which has two steps $t_{3,1}$ and $t_{3,2}$ that read the values of A and B respectively. Thus, these two steps are a transaction ADS segment. Suppose that an application independent scheduling rule R partitions this

transaction ADS segment into two atomic step segments $t_{3,1}$ and $t_{3,2}$ respectively. In the following, we show that such a partition causes transaction T_3 to see an inconsistent state of $\mathcal{A}$.

We construct a schedule $z = \langle T_1, t_{3,1}, T_2, t_{3,2} \rangle$. Note that this schedule satisfies R, since $t_{3,1}$ and $t_{3,2}$ are two different atomic step segments under R. Since T_1 assigns (0, 1) to A and B, step $t_{3,1}$ will see that the value of A is 0. Since T_2 assigns (1, 0) to A and B, step $t_{3,2}$ will see that the value of B is also 0. Thus, under z transaction T_3 sees an inconsistent state (0, 0). Since rule R is application independent, we can interpret the semantics of T_3 as follows. If transaction T_3 sees an inconsistent state, then step $t_{3,2}$ outputs an incorrect value 2. Thus, rule R is incorrect. Since rule R is assumed to be consistent and correct, it cannot further partition transaction ADS segments. It follows that the setwise serializable scheduling rule is at least as concurrent as R. In summary, the key to the proof is to show that if an application independent scheduling rule R partitions a transaction ADS segment σ into two or more atomic step segments, then there exists some schedule z satisfying R such that the inputs to σ under z do not come from a consistent state. We now give a formal proof as follows.

Lemma 4.5-1: Let $\mathcal{A} = \{O_1, \ldots, O_m\}$ be an ADS from the maximal consistency preserving partition of D. Let the index set of $\mathcal{A}$, I, be partitioned into two sets S_1 and S_2. Let U be the universal set of the consistent states of $\mathcal{A}$. We have

$$\exists (W \in U \wedge X \in U \wedge Y \in U)\,((\pi_{s_1}(W) = \pi_{s_1}(Y)) \wedge (\pi_{s_2}(X) = \pi_{s_2}(Y)))$$

Proof: Suppose that the claim is false. By Definition 3.1 (consistency preserving partition), $\{S_1, S_2\}$ is a consistency preserving partition of I. This contradicts the assumption that D is maximally partitioned. □

Lemma 4.5-2: Let database D be maximally partitioned into atomic data sets. Let T_i be a consistent and correct transaction operating upon D. Let σ be a transaction ADS segment in T_i. Let the ADS accessed by σ be $\mathcal{A}$. Suppose that σ is partitioned into atomic step segments $\sigma_1, \ldots, \sigma_k$, $k > 1$ by some modular scheduling rule R. Then there exist a transaction system $T \in \mathbf{T}$ and a schedule $z \in Z_R(T)$ such that under z the values input to σ are not projections from a consistent state of $\mathcal{A}$.

Proof: Let $\Xi_{ADS}(T_i) = \{\sigma_1, \ldots, \sigma_k\}$ be transaction ADS segments of T_i. Suppose that a modular scheduling rule R partitions transaction ADS segment σ_1 into $\sigma_{1,1}, \ldots, \sigma_{1,j}$, $j \geq 2$. In the following, we prove

4.5 Summary

In this chapter, we have defined the setwise serializable scheduling rule and proved that this rule is the optimal application independent scheduling rule. From an application point of view, once the atomic data sets and consistency constraints are specified, the implementation of the setwise serializable scheduling rule is straightforward. For example, we can modify the two phase lock protocol [Eswaran 76] into *setwise two phase lock*. A setwise two phase lock protocol requires a transaction not to release any lock on any data object of an atomic data set until all the locks in this atomic data set have been acquired. Once any lock in an atomic data set has been released, no more data objects in the same atomic data set can be locked. In many instances, data objects in an atomic data set can be organized into a tree structure. In this case, the tree lock protocol [Silberschatz 80] can be adopted to enforce setwise serializability. This protocol has the advantage that it is free from the deadlock problem. This protocol requires that,

- except for the first item locked, no item can be locked unless a lock is currently held on its parent.
- no item is ever locked twice.

Note that the first item need not be the root, and the locking need not to be two phase.

that if j = 2 then the inputs to σ are not projections of a consistent state of $\mathcal{A}$. If j > 2, we merge $\sigma_{1,2}$, ... $\sigma_{1,j}$ into a single atomic step segment σ_2^* and then use the result for j = 2. There can be two cases.

Case 1: Segments $\sigma_{1,1}$ and $\sigma_{1,2}$ access disjoint sets of data objects in ADS $\mathcal{A}$. Let I be the index set of $\mathcal{A}$. Partition I into S_1 and S_2 such that σ_i accesses only data objects indexed by S_i, i = 1 to 2.

Let T_w and T_x be two transactions in T, assigning consistent states W and X to $\mathcal{A}$ respectively. Let $W_1 = \pi_{S_1}(W)$ and $X_2 = \pi_{S_2}(X)$. Although states W and X are consistent, by Lemma 4.5-1 the state resulting from the combination of W_1 and X_2, $Y = \langle W_1, X_2 \rangle$, could be an inconsistent state. We assume that this is the case. Let z be a schedule satisfying R, in which $T_w < \sigma_{1,1} < T_x < \sigma_{1,2}$, where "$T_w < \sigma_{1,1}$" denotes that T_w precedes $\sigma_{1,1}$ in z. Note that the inputs to σ are projections from the inconsistent state Y.

Case 2: Segments $\sigma_{1,1}$ and $\sigma_{1,2}$ share at least one data object in $\mathcal{A}$. Let one of these shared data objects be O. Let z be a schedule satisfying R in which $T_w < \sigma_{1,1} < T_x < \sigma_{1,2}$. Let the values of O assigned by T_w and T_x be w and x respectively. Hence, under z there are two steps accessing O, inputting values from two different states of $\mathcal{A}$. It follows that R cannot guarantee that all the inputs to σ_1 are projections from a consistent state of $\mathcal{A}$. □

Theorem 4.5: Given a maximally CP partitioned database D, the setwise serializable scheduling rule is optimal in the set of application independent scheduling rules for transactions operating upon database D.

Proof: Let the setwise serializable scheduling rule be R^*. Let R be any other application independent scheduling rule. Let T_i be a consistent and correct transaction. Let the database be maximally partitioned into atomic data sets. Let σ be a transaction ADS segment of T_i accessing ADS $\mathcal{A}$. We examine the partition of T_i by R. If any ADS segment σ of T_i is partitioned, then by Lemma 4.5-2 the inputs to σ cannot be guaranteed to be the projections of a consistent state of $\mathcal{A}$. Since R identically partitions all the transactions equivalent to T_i in syntax, we can interpret the semantics of T_i as follows. T_i produces correct results if and only if the input values to σ are projections from a consistent state of $\mathcal{A}$. Hence, if any transaction ADS segment is partitioned by R, we can construct a consistent and correct transaction T_i and a schedule z satisfying R such that under z, T_i produces incorrect results. Thus, R is incorrect. It follows that the atomic step segments of T_i specified by R can only be either transaction ADS segments or the supersets of them. Therefore, any schedule z satisfying R satisfies R^*. Hence, R^* is at least as concurrent as R. □

5. Compound Transactions

In the previous chapter, we studied the optimal application independent scheduling rule – the setwise serializable scheduling rule. We now search for an even higher degree of concurrency in modular concurrency control by removing the requirement of application independence. This objective is achieved by developing a new transaction structure called *compound transactions*, and its associated scheduling rules called *generalized setwise serializable* scheduling rules. The structure of a compound transaction is designed to utilize the semantic information of a given transaction by partitioning its steps into a partially ordered set of *elementary transactions*. The syntax of each elementary transaction can be in the form of either a single level or a nested transaction. A compound transaction becomes a nested transaction when it consists of only a single elementary transaction.

Conceptually, while atomic data sets result from a consistency preserving partition of the database, elementary transactions result from a correctness preserving partition of a transaction. Given a database and its associated consistency constraints, we partition the database into consistency preserving atomic data sets. As long as the consistency of each atomic data set is satisfied individually, the consistency of the database is also satisfied. Given a transaction and its associated post-conditions, we partition the transaction into correctness preserving elementary transactions. As long as the post-condition of each elementary transaction is individually satisfied, the post-conditions of the compound transaction are also satisfied. More precisely, a compound transaction consists of a partially ordered set of elementary transactions. When an elementary transaction is executed serializably and in an order consistent with the partial ordering of the elementary transactions in the compound transaction, it has the following two properties. First, given a consistent state of the database and the results passed from preceding elementary transactions, an elementary transaction produces another consistent state of the database and satisfies its own post-conditions. Second, the conjunction of the post-conditions of the constituent elementary transactions is equivalent to the post-conditions of the compound transactions. A simple example of compound transaction was given in the overview of this thesis. A more elaborated example is given in Section 7.3, Writing Compound Transactions.

We now turn to the subject of scheduling compound transactions. Given a transaction, the *generalized setwise serializable* rules call for the following two steps. First, we need to express the transaction in the form of a compound transaction. Second, we partition each of the elementary transactions in the compound

transaction into transaction ADS segments. We have proved that generalized setwise serializable scheduling rules are consistent, correct and modular. In addition, these rules form a complete class within the set of all the possible modular scheduling rules. This means that for any given modular scheduling rule, there exists a generalized setwise serializable scheduling rule that provides at least as much concurrency.

This chapter is organized as follows. In Section 5.1, we define the syntax of compound transactions and state our assumptions about them. In Section 5.2, we define generalized setwise serializable scheduling rules and show that they are consistent, correct and modular. In addition we show that generalized setwise serializable scheduling rules form a complete class within the set of all the consistent and correct modular scheduling rules. Section 5.3 is the summary of this chapter.

5.1 Syntax

We begin our formal investigation of compound transactions by first defining their syntax. A compound transaction consists of a partially ordered set of elementary transactions. Each elementary transaction can have the structure of a nested transaction. These elementary transactions collectively carry out the task of the compound transaction by passing information via local variables. For simplicity, we assume that all elementary transactions are single level transactions in the following discussion. The extension of the results to elementary transactions in the form of nested transactions is straightforward. It can be done via the concept of linearization, similar to the procedure used in Section 4.3.

Definition 5.1: A compound transaction is a partially ordered set of elementary transactions.

Definition 5.2: Let $T_i^e = \{t_{i,1}, ..., t_{i,k}\}$ be an elementary transaction. Let the data object accessed by step $t_{i,j} \in T_i^e$ be $O_{t_{i,j}}$. Let the local variable associated with step $t_{i,j}$ be $L_{t_{i,j}}$. Step $t_{i,j}$ is modelled by the indivisible operation of the following two instructions.

$$L_{t_{i,j}} := O_{t_{i,j}}$$

$$O_{t_{i,j}} = f_{t_{i,j}}(L_p, L_{t_{i,1}}, ..., L_{t_{i,j}})$$

where L_p is the set of local variables associated with all the preceding elementary transactions in the same compound transaction.

We now define the concepts of input and output steps of an elementary transaction. Next, we define the concept of the post-conditions of an elementary transaction.

Definition 5.3: Step $t_{ij} \in T_i^e$ is an input step if it is the step in T_i^e first accessing a data object O. Step t_{ij} is an output step if it is the step in T_i^e last accessing O.

Definition 5.4: Let $O_j[v_b]$, j = 1 to k, be the values input to the input steps of T_i^e and $O_j[v_f]$, j = 1 to k, be the values output by the output steps of T_i^e. The post-condition of T_i^e is a specification of the output values as functions of input values and the values of local variables associated with the steps in the preceding elementary transactions.

$$O_j[v_f] = f_j(L_p, O_1[v_b], ..., O_k[v_b]), j = 1 \text{ to } k.$$

where L_p is the set of local variables associated with the steps in the preceding elementary transactions of the same compound transaction.

Having defined the syntax of compound transactions, we now state our assumptions about them.

Assumption 5.1: When an elementary transaction of a compound transaction is executed serially and in an order that is consistent with the partial order defined by the compound transaction, it satisfies our three fundamental assumptions about a transaction, that is, it terminates (A1), preserves the consistency of the database (A2) and satisfies its own post-conditions (A3). Furthermore, the post-condition of a compound transaction is equivalent to the conjunction of all the post-conditions of its elementary transactions.

5.2 Generalized Setwise Serializable Scheduling Rules

Having defined the syntax of a compound transaction, we now define the concept of generalized setwise serializable scheduling rules.

Definition 5.5 Let $T = \{T_1, ..., T_n\}$ be a transaction system. The generalized setwise serializable scheduling rule R is a modular scheduling rule with the following properties.

1. $R(T_1, ... T_n) = (R_1^1(T_1), ..., R_1^n(T_n))$

2. The kernel of R, R_1^i, i = 1 to n, is a composite function which first maps T_i into a compound

transaction T_i^c, and then partitions the steps of each of the elementary transactions of T_i^c into transaction AIDS segments.

For simplicity, we refer to the kernels of the generalized setwise serializable scheduling rule as generalized setwise serializable scheduling rules.

Definition 5.6: A schedule z for a transaction system T is said to be generalized setwise serializable if z satisfies the generalized setwise serializable scheduling rule R. That is, the transaction AIDS segments specified by R in one elementary transaction are interleaved serializably with those of others in z.

Theorem 5.1: Generalized setwise serializable schedules are consistent and correct.

Proof: Since a generalized setwise serializable schedule is setwise serializable with respect to all the elementary transactions in the system, it follows from Assumption 5.1 and Corollary 4.2-1 that each elementary transaction terminates, preserves the consistency of the database and produces results that satisfy its post-conditions. Hence, the consistency of the database is preserved and the post-condition of each of the compound transactions is also satisfied. Therefore, generalized setwise serializable schedules are consistent and correct. □

Corollary 5.1: Generalized setwise serializable scheduling rules are consistent and correct.

Theorem 5.2: Generalized setwise serializable scheduling rules are modular.

Proof: It directly follows from Definition 5.5. □

Thus far, we have shown that the generalized setwise serializable scheduling rule R is consistent, correct and modular. We now prove that generalized setwise scheduling rules (i.e. the kernels of R) form a complete class within the set of all the consistent and correct modular scheduling rules. Before proceeding with the proof of completeness, we need to introduce the concept of the post-conditions associated with an atomic step segment. To illustrate the need, let the transaction $T_i = \{t_{i,1}: A := A - 1; t_{i,2}: A := A + 1\}$. If the two steps of T_i are treated as a single atomic step segment, then $t_{i,1}$ is an input step and $t_{i,2}$ is an output step. The partition of a transaction could create input and output steps in addition to those defined in an executing alone environment. For example, if each of these two steps is an atomic step segment, then $t_{i,1}$ ($t_{i,2}$) is both an input step and an output step.

Definition 5.7: Let $\sigma = \{t_{i,1}, ..., t_{i,k}\}$ be an atomic step segment. Let the data object accessed by step $t_{i,j} \in \sigma$ be O. Step $t_{i,j}$ is an input step if it is the step in σ first accessing O. Step $t_{i,j}$ is an output step if it is the step in σ last accessing O.

Definition 5.8: Let $O_j[v_i]$, j = 1 to k, be the values input to the input steps of σ and $O_j[v_f]$, j = 1 to k, be the values output by the output steps of σ. The post-condition of σ is a specification of the output values as functions of input values and the values of the local variables associated with the steps of preceding atomic step segments of the same transaction. That is,

$$O_j[v_f] = f_j(L_j, O_1[v_b], ..., O_k[v_b]), j = 1 \text{ to } k;$$

where L_j is the set of local variables associated with the steps of the atomic step segments preceding the output step for O_j.

We now prove that generalized setwise serializable scheduling rules form a complete class within the set of all the consistent and correct modular scheduling rules.

Lemma 5.3: Let R be a modular scheduling rule. R is consistent and correct if and only if

1. for each of the transactions T_i in T, the conjunction of the post-conditions of all the atomic step segments in T_i is equivalent to the post-condition associated with T_i.

2. let z be a schedule for a transaction system $T \subset \mathbb{T}$ and z satisfies R. In the execution of T according to z, any atomic step segment in $T_i \in T$ specified by R preserves the consistency of the database.

Proof: First, if any atomic step segment σ in T_i specified by R does not preserve the consistency of the database, then another transaction T_j executing after σ would input an inconsistent state. Since R is modular, we can define the semantics of T_j as one that outputs incorrect results when its input is inconsistent. Thus R is incorrect. Second, if the conjunction of the post-conditions of all the atomic step segments in T_i specified by R is not equivalent to the post-condition associated with T_i, then R is incorrect by definition. Since any schedule z for any transaction system $T \subset \mathbb{T}$ satisfying R guarantees the serializability in the execution of the transaction atomic step segments specified by R, it follows from conditions 1 and 2 that z is consistent and correct. □

Theorem 5.3: Generalized setwise serializable scheduling rules form a *complete class* within the set of modular scheduling rules.

Proof: Let R be a consistent and correct modular scheduling rule. Suppose that T_i is a consistent and correct transaction and T_i is partitioned into atomic step segments $\sigma_1, \ldots, \sigma_k$ by R. First, by Lemma 5.3 σ_i, i = 1 to k, must preserve the consistency of the database when executing alone. Second, by Lemma 5.3 the conjunction of the post-conditions of $\sigma_1, \ldots, \sigma_k$ must be equivalent to the post-conditions associated with T_i. Note that each σ_i, i = 1 to k, satisfies the definition of an elementary transaction. We now define a generalized setwise serializable scheduling rule R^* which partitions T_i as follows. First, R^* labels $\sigma_1, \ldots, \sigma_k$ as elementary transactions. Next, R^* partitions these elementary transactions into transaction ADS segments. Hence, R^* is at least as concurrent as R. □

5.3 Summary

In this chapter, we have defined the concept of compound transactions and their associated scheduling rules: the generalized setwise serializable scheduling rules. We have proved that these rules are consistent, correct and modular. In addition, generalized setwise serializable scheduling rules form a complete class within the set of all the consistent and correct modular scheduling rules.

From an application point of view, it is important to point out that the process of mapping a transaction into the form of a compound transaction requires the understanding of the semantics of the transaction. We have only provided the syntax of compound transactions to allow a user to accomplish this process. It is the user's responsibility to prove that he has correctly expressed his transaction in the form of a compound transaction. The scheduling of compound transactions is, however, straightforward. We can use either setwise two phase lock or tree lock to ensure that each elementary transaction in a compound transaction is executed setwise serializably.

From a programmer's point of view, the implication of our completeness result, Theorem 5.3, is that there exists such a way of writing compound transactions that we can provide at least as much concurrency as any other modular concurrency control method. Unfortunately, this is only an existence theorem. In order to achieve a high degree of concurrency, we must try to make a compound transaction consisting of many small elementary transactions rather than a few large ones. Some of the ideas in writing and scheduling of a compound transaction is illustrated by an example in Section 7.3, Writing Compound Transactions.

6. The Failure Safe Rule

In the previous chapters, we have studied the subject of concurrency control without considering the impact of possible system failures. We now extend our work by taking the effect of system failures into consideration. Computer systems can fail in many ways. We limit ourself to the so-called *clean and soft* failures that are traditionally considered by transaction system failure recovery management [Bernstein 83]. A failure is said to be clean and soft if the effects of this failure can be modelled by a network of computers some of which stop running and lose the content of their main memories. However, the database residing in the stable storage remains intact. The stable storage is an idealized model of a network of system disks or other reliable storage devices whose probability of failure is so small that such failures can be ignored by the applications in question.

The objective of using failure recovery rules is to ensure that concurrency control can be carried out consistently and correctly despite failures in the system. A failure recovery rule specifies the conditions under which the executed steps of a transaction commit or abort. Committing a sequence of executed steps means non-divisibly transferring the result of computations from the main memory to the database residing in the stable storage. The commit operation represents irrevocable changes made to the database. For example, the recovery rule associated with serializable schedules is known as *failure atomicity*. This rule requires that a transaction either commits or aborts all executed steps at the end of execution. The non-divisibility for committing all the executed steps of a transaction is a main cause of low concurrency. In order to ensure failure atomicity and avoid cascaded aborts, it was shown by Wood [Wood 80] that none of the locks held by a transaction can be released until the transaction has successfully committed. This is because a transaction can be aborted by failures before it has committed. To release a lock before a transaction has committed could allow other transactions to use the results of a partial computation that could be invalidated by a later abort operation. Cascaded abort is generally regarded as a very undesirable event and should be avoided. In this work, we consider only recovery rules for which cascaded aborts do not occur.

In this chapter, we develop a new failure recovery rule, the *failure safe* rule, which is designed for transaction systems using generalized setwise serializable schedules. Given a compound transaction, the failure safe rule examines the transaction ADS segments in each of the elementary transactions. If the steps of a transaction ADS segment are not interleaved with those of others, then the ADS segment is named an *atomic commit segment*. Otherwise, interleaved ADS segments are taken to be a single atomic commit segment. That

is, the failure safe rule partitions a transaction into a partially ordered set of atomic commit segments. A transaction must non-divisibly commit each of its atomic commit segments in an order consistent with the partial order of commit segments in the transaction. As an example, the compound transaction Get-A-and-B discussed in the overview of this thesis has four atomic commit segments, each of which corresponds to one of the four elementary transactions: Get-A, Get-B, Put-Back-A and Put-Back-B, because each of these elementary transactions consists of a single transaction ADS segment. It should be pointed out that failure atomicity is a degenerate case of our failure safe rule in that it corresponds to the rule in which the entire set of steps of a transaction is taken as a single atomic commit segment.

From a concurrency control point of view, a concurrent execution of a transaction system is modelled by a schedule. It is shown in this chapter that owing to the preservation of both the internal ordering of steps in transactions and the integrity of transaction ADS segments, the effect of a system failure under the failure safe rule becomes equivalent to changing an existing generalized setwise serializable schedule to another schedule that is also generalized setwise serializable. Thus, under the failure safe rule system failures are *safe* in the sense that the computations recorded in the database are equivalent to the computations resulting from a failure-free execution of the transaction system.

In this chapter, the organization of studying failure recovery rules is parallel to that of our studying concurrency control rules in the previous chapters. In Section 6.1, we develop a model of failure recovery rules and define their properties such as consistency, correctness failure safety and optimality. In Section 6.2 we define the failure safe rule and prove that this rule is consistent, correct and modular. In addition, we prove that this rule is optimal for transaction systems scheduled by generalized setwise serializable scheduling rules. In Section 6.3, we summarize this chapter.

6.1 A Model of Modular Failure Recovery Rules

We now develop our model of modular failure recovery rules. We first define the concept of a modular failure recovery rule. Second, we model the effect of clean and soft failures upon concurrency control. We then define the concept of consistency and correctness for a modular failure recovery rule in the face of clean and soft failures. Next, we introduce the concept of safety for a recovery rule. We conclude this section by addressing the issues of re-scheduling aborted transactions.

A failure recovery rule is a function which takes a transaction and partitions it into equivalent classes called *atomic commit segments*. At the end of executing an atomic commit segment, all executed steps of this segment must be either committed or aborted as an indivisible unit. A failure recovery rule is said to be modular if it partitions a transaction independently of other transactions in the system. For example, failure atomicity is a modular failure recovery rule that takes the entire set of steps of a transaction as a single atomic commit segment. We now formalize our concept of a failure recovery rule in Definitions 6.1-1 and 6.1-2.

Definition 6.1-1: Let $\mathcal{T}_m$ denote the set of all the possible consistent and correct transactions with m steps. Let $\mathcal{T}$ denote the set of all the consistent and correct transactions, that is, $\mathcal{T} = \cup_{m=1}^{\infty} \mathcal{T}_m$. Let P_m denote a partition of an m-step consistent and correct transaction into a partially ordered set of transaction step segments called atomic commit segments. Let $\mathcal{P}_m$ denote the set of all the possible sets of partitions of an m-step consistent and correct transaction. Let $\mathcal{P}$ be the set of all the possible partially ordered sets of partitions, that is, $\mathcal{P} = \cup_{m=1}^{\infty} \mathcal{P}_m$. A failure recovery rule for a transaction system with n transactions, $\mathcal{R}_n$, is a function which takes the transaction system of size n and partitions each of the transactions into a partially ordered set of atomic commit segments.

$$\mathcal{R}_n: \prod_{i=1}^{n} \mathcal{T} \rightarrow \mathcal{P}$$

Definition 6.1-2: A failure recovery rule $\mathcal{R}$ is a function which takes a transaction system of any size and partitions each of the transactions into a partially ordered set of commit segments.

$$\mathcal{R}: \cup_{n=1}^{\infty}(\prod_{i=1}^{n} \mathcal{T}) \rightarrow \cup_{n=1}^{\infty}(\prod_{i=1}^{n} \mathcal{P})$$

such that the restriction of $\mathcal{R}$ to $\prod_{i=1}^{n} \mathcal{T}$ is $\mathcal{R}_n$, i.e.

$$\mathcal{R}|\prod_{i=1}^{n} \mathcal{T} = \mathcal{R}_n, \; n = 1 \text{ to } \infty.$$

Definition 6.2: A failure recovery rule $\mathcal{R}$ is said to be *modular* if and only if $\mathcal{R}$ independently partitions each transaction in a transaction system into atomic commit segments. That is,

$$\mathcal{R}_n(T_1, ..., T_n) = (\mathcal{R}_1(T_1), ..., \mathcal{R}_1(T_n)), \; n = 1 \text{ to } \infty,$$

where $\mathcal{R}_n$ is the restriction of $\mathcal{R}$ to $\prod_{i=1}^{n} \mathcal{T}$.

Having formalized the concept of modular recovery rules, we now state our assumption regarding the atomic commit segments produced by those recovery rules. When failure atomicity is adopted, the commit operation ensures that the computations produced by a transaction are either discarded or transferred to the database as a non-divisible unit. In this case, the values stored in the local variables are irrelevant to the commit operation, because the computation has been completed. However, when we commit a transaction segment by segment, we must not only guarantee that the computations produced by an atomic commit segment are non-divisibly transferred to the database, but also guarantee that the values stored in the local variables associated with the commit segment are non-divisibly transferred to the stable storage as well. This is because when an aborted transaction resumes its execution, an executing step may need the values of the local variables associated with those commit segments already committed. We now formalize our assumption regarding the commit and abort operations of atomic commit segments.

Assumption 6.1-1: At the end of executing an atomic commit segment, the computations produced by this segment will be either *committed* or *aborted*.

Assumption 6.1-2: An atomic commit segment σ is said to be *committed* if and only if

1. the computations produced by this segment will be non-divisibly transferred to the database. That is, let the data objects accessed by atomic commit segment σ be $O_1, ..., O_k$ and the values output by σ be $O_1[v_f], ..., O_k[v_f]$. The values of the data objects in the database will be $O_1[v_f], ..., O_k[v_f]$ if σ is committed.

2. The values stored in the local variables associated with segment σ will also be non-divisibly transferred to the stable storage. That is, let $L_1, ..., L_k$ be the set of local variables associated with the transaction steps in σ. Let $\mathcal{L}_1, ..., \mathcal{L}_k$ be a set of data objects in the stable storage but not part of the database. Data objects $\mathcal{L}_1, ..., \mathcal{L}_k$ are said to be the *private stable storage* for σ. The private stable storage of an atomic commit segment σ can only be written by the commit operation of σ and can only be read by steps of other commit segments in the same transaction. When σ has committed, we have $\mathcal{L}_1 = L_1, ..., \mathcal{L}_k = L_k$.

Assumption 6.1-3: An atomic commit segment is said to be aborted if and only if its computations are discarded. That is,

1. Let $O_1[v_b], ..., O_k[v_b]$ be the values of data objects $O_1, ..., O_k$ input to atomic commit segment σ. The values of data objects $O_1, ..., O_k$ remain to be $O_1[v_b], ..., O_k[v_b]$ respectively.

2. The private stable storage associated with σ has the initial value "nil" for each of the data objects $\mathcal{L}_1, ..., \mathcal{L}_k$. The value "nil" is special value reserved for recovery management such as the bit

pattern of a word with all 1's. "nil" must not be in the domain of any data object. When the private stable storage $\mathcal{L}_1, \ldots, \mathcal{L}_k$ are assigned to a commit segment, their initial values are initialized to "nil's". This indicates that they have not been used to store values by the commit segment.

Assumption 6.1-4: When resuming the execution of an aborted atomic commit segment $\sigma \in T_i$, all the local variables associated with *committed* atomic commit segments in T_i will be restored to values saved in their private stable storages. That is, let $\sigma_1, \ldots, \sigma_k$ be the atomic commit segments that have been committed. We have

$$\forall (L_i \in \sigma_j, 1 \leq j \leq k)\,(L_i = \mathcal{L}_i)$$

Having defined modular recovery rules and the commit operations for atomic commit segments, we now model the effect of a clean and soft system failure upon concurrency control. When a transaction system is executed according to a schedule z and a failure occurs, schedule z is partitioned into two parts. The part z^e represents the steps in z that have been executed, another part z^f represents those steps yet to be executed.[1]

Definition 6.3: Given a schedule z for transaction system T, a clean and soft system failure partitions z into two parts: z^e and z^f. Partial schedule z^e represents steps of z that have been executed prior to the failure and z^f represents steps in z yet to be executed.

Within an executed partial schedule z^e, there can be some atomic commit segments which have been committed. The computations represented by successfully committed atomic commit segments is modelled by a *committed partial schedule* z^c.

Before we formally define this concept, we need first to introduce a useful concept called *sub-segment*, denoted as "$\sqsubseteq$". We say that a sequence of steps σ_2 is a sub-segment of another sequence of steps σ_1 if and only if all the steps in σ_2 are also in σ_1. In addition, the ordering of steps in σ_2 is consistent with that in σ_1.

Definition 6.4: Let σ_1 be a sequence of transaction steps. A sequence of steps σ_2 is said to be a sub-segment of σ_1, denoted as "$\sigma_2 \sqsubseteq \sigma_1$", if and only if

---

[1] It is important to recall that each step is assumed to be executed atomically. Thus, we do not need to deal with a "partially executed" step in our model.

$$\forall((t_i,t_j \in \sigma_2,\ i \neq j) \wedge (t_i > t_j))((t_i,t_j \in \sigma_1) \wedge (t_i > t_j))$$

We now define the concept of a committed partial schedule.

Definition 6.5: Let z^e be the executed partial schedule in z. Let σ be an atomic commit segment. The *committed partial schedule* z^c is a sub-segment of z^e that contains only successfully committed atomic commit segments. That is,

1. The committed partial schedule is a sub-segment of the executed partial schedule.

   $z^c \sqsubseteq z^e$;

2. The committed partial schedule does not contain fragmented atomic commit segments.

   $\forall(t \in z^c)(\exists(\sigma \sqsubseteq z^c)(t \in \sigma))$

3. Transaction commit segments belonging to a transaction are committed in the order defined by the commit rule. We let "$\sigma_1 < \sigma_2$" denote that commit segment σ_1 precedes σ_2.

   $$\forall(T_i \in T)\forall((\sigma_1,\sigma_2 \sqsubseteq T_i) \wedge (\sigma_1 < \sigma_2))((\sigma_2 \sqsubseteq z^c) \rightarrow (\sigma_1 \sqsubseteq z^c))$$

Having addressed the issues related to the commit operation, we now turn to the subject of resuming the execution of an aborted transaction. When a transaction fails, some of its atomic commit segments may have been already committed. When the database system resumes the execution of this aborted transaction, we are facing the problem of scheduling a partial transaction. We assume that a partial transaction is scheduled by the same scheduling rule R as follows. When a partial transaction consists of an integer number of atomic step segments, all we have to do is to ensure that these segments are interleaved serializably with those of other transactions. The main problem in scheduling a partial transaction is that an atomic step segment specified by R might be partitioned by a recovery rule $\mathfrak{R}$ into more than one atomic commit segments. When a failure occurs, it is possible that only some of these commit segments have been committed. In this case, the portion of an atomic step segment left in the remaining schedule z_1 will be taken as an atomic step segment and interleaved serializably with atomic step segments of other transactions in z_1.

For example, suppose that transaction T_i has two atomic step segments σ_1 and σ_2 specified by scheduling rule R. If the entire T_i is in a remaining partial schedule z_1, then these two atomic step segments will be interleaved serializably with atomic step segments of other transactions in z_1. If σ_1 is committed but σ_2 is left

in z_1, then σ_2 will be interleaved serializably with those of others in z_1. Finally, suppose that a recovery rule partitions the atomic step segment σ_1 into two atomic commit segments: $\sigma_{1.1}$ and $\sigma_{1.2}$. If only $\sigma_{1.1}$ has been committed and $\sigma_{1.2}$ and σ_2 are left in z_1, then both $\sigma_{1.2}$ and σ_2 will be taken as atomic step segments in z_1. That is, $\sigma_{1.2}$ and σ_2 will be interleaved serializably with those of others in z_1. We formalize our discussion about scheduling aborted transactions as Assumption 6.2.

Assumption 6.2: Let z be a schedule for transaction system T satisfying scheduling rule R. Let z_1 be the remaining partial schedule after a failure occurs during the execution of z. Let $\Xi_R(T_i)$ denote the atomic step segments of transaction T_i specified by R.

1. The steps in z_1 are those in z but not in the committed partial schedule z^c.

   $$\forall(t)((t\in z)\wedge(t\notin z^c))\rightarrow(t\in z_1))$$

2. If all the steps of a transaction T_i is in z_1, then the atomic step segments of T_i are still represented by $\Xi_R(T_i)$.

3. Let T_i^p denote the partial transaction of T_i in the remaining partial schedule z_1. Partial transaction T_i^p is a sequence of steps such that $(T_i^p\sqsubseteq T_i)\wedge(\forall(t)\,(\,((t\in T_i)\wedge(t\in z_1))\rightarrow(t\in T_i^p)\,)$. Partial transaction T_i^p is scheduled by R as follows. Let $\Xi_R(T_i^p)$ denote the atomic step segments specified by R with respect to the partial transaction T_i^p. We have

   a. $\forall(T_i^p\sqsubseteq z_1)\forall(\sigma\in\Xi_R(T_i^p))\,(\,\exists(\sigma^*\in\Xi_R(T_i))(\sigma\sqsubseteq\sigma^*)\,)$

   b. $\forall(\sigma^*\in\Xi_R(T_i))\;\forall((t\in\sigma^*)\wedge(t\in T_i^p))\,(\,\exists(\sigma\in\Xi_R(T_i^p))(t\in\sigma))$

4. The remaining partial schedule z_1 satisfies scheduling rule R. That is, the atomic step segments specified by R in a (partial) transaction will be interleaved serializably with those of others in z_1.

Having modelled the effect of a single failure, we now address the issue of multiple failures. When the first failure occurs during the execution of z, the committed atomic commit segments are represented by z^c. The executed but not yet committed transaction steps are aborted. These aborted steps are re-scheduled together with steps in z^f to create the *remaining partial schedule* z_1. In the execution of z_1, suppose that the second failure occurs. In this case, we have a new committed partial schedule, z_1^c, and a new remaining schedule z_2. This process continues until all the steps in the the transaction system are executed and committed.

Definition 6.6-1: Let z be a schedule of transaction system T satisfying scheduling rule R. When the first

failure occurs, the committed partial schedule is denoted as z^c and the remaining partial schedule is denoted as z_1. The k^{th} remaining partial schedule z_k is the remaining schedule for z_{k-1} after the k^{th} failure.

Definition 6.6-2: When there are no failures, the execution of a transaction system is modelled by a scheduling z, with n failures the execution of a transaction system is modelled by a committed schedule $z(n) = (z^c, z_1^c, ... z_n^c)$.

In order to investigate the execution of transaction systems in the face of failures, we must relate the concept of a committed schedule to our established results of scheduling rules.

Theorem 6.1: Let a committed schedule for a given total of k failures in the execution of transaction system $T = \{T_1, ... T_n\}$ be $z(k) = \langle z^c, z_1^c, ... z_k^c \rangle$, where z_i^c is the i^{th} committed partial schedule after the i^{th} failure.[2] Committed schedule $z(k)$ satisfies scheduling rule R if and only if

1. The ordering of the steps of transaction T_i, $1 \leq i \leq n$, in $z(k)$ is consistent with that in transaction T_i, $1 \leq i \leq n$.

$$[\forall (t)((t \in z(k)) \Leftrightarrow (t \in \cup T))] \wedge [\forall (T_i \in T) \forall ((t_{i,j}, t_{i,k} \in T_i) \wedge (t_{i,k} > t_{i,j})) ((t_{i,j}, t_{i,k} \in z(k)) \wedge (t_{i,k} > t_{i,j}))]$$

2. Atomic step segments specified by R and belonging to different transactions are interleaved serializably in z.

$$\exists (z \in Z(T)) ((z(k) \equiv z) \wedge (z \text{ is atomic step segment serial}))$$

Proof: It directly follows from Assumptions 6.1-1, 6.1-2, 6.1-3 and 6.1-4, Definitions 6.6-1 and 6.6-2, and Definitions 2.10 and 2.16. □

Having modelled the effect of failures upon concurrency control, we are now in a position to define the concept of consistency and correctness of a recovery rule. We consider a recovery rule $\mathfrak{R}$ designed for a consistent and correct scheduling rule R is consistent and and correct if and only if $\mathfrak{R}$ ensures the consistency and correctness of the committed schedules in the face of system failures.

Definition 6.7: Let $\mathfrak{R}$ be the recovery rule designed for the scheduling rule R, recovery rule $\mathfrak{R}$ is said to be consistent and correct if and only if

---

[2] Note that $z_k^c = z_k$ because there are no more failures after the k^{th} failure and the entire z_k is committed.

$\forall (z \in Z_R(T))\ (z(n)$ is consistent and correct, $0 \leq n \leq \infty)$

where $Z_R(T)$ is the set of all the schedules for transaction system T satisfying R.

We now define an important property of a recovery rule called *safety*. We consider a recovery rule $\mathfrak{R}$ designed for scheduling rule R as being *safe* if and only if the committed schedules satisfy scheduling rule R. That is, the computations recorded in the database cannot be distinguished from those resulting from an execution of a schedule $z \in Z_R(T)$ without failures.

Definition 6.8: A recovery rule $\mathfrak{R}$ designed for a scheduling rule R is said to be *safe* if and only if for any given schedule $z \in Z_R(T)$ the commit schedules satisfy R, i.e.

$\forall (z \in Z_R(T))(z(n) \in Z_R(T), 0 \leq n \leq \infty)$

Theorem 6.2: If recovery rule $\mathfrak{R}$ designed for a consistent and correct scheduling rule R is safe, then recovery rule $\mathfrak{R}$ is consistent and correct.

Proof: Since all the schedules satisfying scheduling rule R are consistent and correct, it follows from Definitions 6.7 and 6.8 that $\mathfrak{R}$ is consistent and correct. □

We now turn to the subject of optimality of modular and safe recovery rules. We measure the concurrency provided by a modular recovery rule by how fine it partitions a transaction. This is because computations produced by an atomic commit segment must be withheld by locks or other mechanisms until this segment has been committed. To allow other transactions using the results produced by a commit segment before its commit could lead to cascaded aborts – a highly undesirable event. Generally, the finer the partition, the smaller is the size of a commit segment and thus the higher the degree of concurrency.

Definition 6.9: Let $\mathbb{T}$ be the set of all the consistent and correct transactions. Let $\mathbb{R}$ be the set of all the modular and safe recovery rules associated with some consistent and correct modular scheduling rule R. Let $\Xi_{\mathfrak{R}}(T_i)$ denote the partition resulting from $\mathfrak{R} \in \mathbb{R}$ partitioning $T_i \in \mathbb{T}$. Modular and safe recovery rule $\mathfrak{R}$ is said to be optimal with respect to the associated scheduling rule R, if and only if $\mathfrak{R}$ always produces the most refined partitions. That is, $\mathfrak{R}$ is optimal if and only if

$$\forall (T_i \in T) \forall (\mathcal{R}^* \in \mathbb{R}) ((\sigma \in \Xi_{\mathcal{R}}(T_i)) \rightarrow (\sigma \in \Xi_{\mathcal{R}^*}(T_i))).$$

where σ denotes an atomic commit segment.

We now conclude this section by addressing the issues in re-scheduling those aborted transactions. The main point is that an atomic commit segment must be a superset of atomic step segments.

Theorem 6.3: An atomic commit segments produced by a modular and safe recovery rule must be a superset of atomic step segments produced by the associated scheduling rule R.

Proof: Suppose that this claim is false and there exists a modular and safe recovery rule $\mathcal{R}^*$, which divides atomic step segment σ_1 of transaction T_1 into n commit segments with $n \geq 2$. Let the commit segments be $\langle \sigma_{1.1}, \ldots, \sigma_{1.n} \rangle$. Let the set of data objects read or written by σ_1 be $\mathcal{A}$. Let transaction T_2 consists of only one atomic step segment σ_2 which writes into every data object in $\mathcal{A}$. Now consider a schedule z for transaction system $T = \{T_1, T_2\}$. Suppose that schedule z satisfies R and we execute T according to z in which T_2 is executed last. Suppose that a failure occurs just after $\sigma_{1.1}$ has been committed and commit segments $\sigma_{1.2}, \ldots \sigma_{1.n}$ are aborted. Let the remaining partial schedule z_1 be $\langle T_2, \sigma_{1.2}, \ldots, \sigma_{1.n} \rangle$. Suppose that there are no more failures. That is, $z(1)$ is the committed schedule. Note that $z(1)$ does not satisfy R. This is because in $z(1) = \langle \sigma_{1.1}, \sigma_2, \sigma_{1.2}, \ldots, \sigma_{1.n} \rangle$, σ_1 precedes σ_2 but σ_2 also precedes σ_1 on $\mathcal{A}$. That is, the atomic step segments of T_1 are not interleaved serializably with that of T_2. This contradicts the assumption that $\mathcal{R}^*$ is safe. □

6.2 The Failure Safe Rule

Having developed a model of modular recovery rules, we now define our failure safe rule and prove that this rule is modular, consistent, correct and safe. We conclude this section by showing that this rule is optimal for transaction systems using generalized setwise serializable schedules.

When we study schedules, a useful concept is the serializability of interleaving the atomic step segments of one transaction to those of other transactions. In the study of failure recovery, an important concept is the interleaving of transaction ADS segments within the same transaction. When transaction ADS segments of a given transaction are interleaved, they must be committed as a single atomic commit segment in order to preserve the internal ordering of steps in a transaction.

Definition 6.10: Let $\Xi_g(T_i)$ denote the set of transaction ADS segments resulting from a generalized setwise scheduling rule R_g partitioning T_i. A transaction ADS segment $\sigma_i \in \Xi_g(T_i)$ is said to be *interleaved* with transaction ADS segment $\sigma_j \in \Xi_g(T_j)$ if

$$\exists (t \in \sigma_i)(t^1_{\sigma_j} < t < t^m_{\sigma_j})$$

where $t^1_{\sigma_j}$ and $t^m_{\sigma_j}$ are the first and last steps in segment σ_j respectively.

Definition 6.11: Given a transaction $T_i \in \mathcal{T}$ scheduled by a generalized setwise serializable scheduling rule R_g, the *failure safe rule* $\mathcal{R}_f$ is a function that produces a partially ordered set of atomic commit segments as follows:

1. For each transaction ADS segment σ in an elementary transaction of T_i, name σ as an atomic commit segment if σ is not interleaved with any other transaction ADS segment in this elementary transaction.

2. When two or more transaction ADS segments in an elementary transaction are interleaved with each other, name them as a single atomic commit segment.

Theorem 6.4: The failure safe rule is modular.

Proof: It directly follows from Definitions 6.2 and 6.11. □

Theorem 6.5: The failure safe rule $\mathcal{R}_f$ is safe. That is, under $\mathcal{R}_f$ we have,

$$\forall (T \subset \mathcal{T}) \forall (z \in Z_{R_g}(T))\ (z(n) \text{ is generalized setwise serializable, } 0 \leq n \leq \infty):$$

where $z(n) = \langle z^c, \ldots z^c_n \rangle$ is a committed schedule for a given total of n failures.

Proof: By Theorem 6.1, to show that $z(n) = \langle z^c, \ldots, z^c_n \rangle$ is generalized setwise serializable for $0 \leq n \leq \infty$, we need to prove that the ordering of steps in $z(n)$ is consistent with the internal ordering of steps in each of the compound transactions. In addition, all the transaction ADS segments in an elementary transaction must be interleaved serializably with those of others in $z(n)$.

To prove that the internal ordering of transaction steps in a compound transaction is preserved , we need to prove that the ordering of the elementary transactions of a compound transaction in $z(n)$ is consistent with

that in the compound transaction and that the internal ordering of steps in each of the elementary transactions is preserved. By Definition 6.11, the partial ordering of atomic commit segments is consistent with the partial ordering of elementary transactions in a compound transaction. Since atomic commit segments are committed in an order consistent with the partial ordering of atomic commit segments, the ordering of elementary transactions of a compound transaction in $z(n)$ is consistent with the ordering of them in the compound transaction.

To complete the proof that the internal ordering of steps in a compound transaction is preserved, we need to show that the ordering of steps in each of the elementary transactions is also preserved. Let $t_{i,k}$ and $t_{i,m}$ be two steps in an elementary transaction T_i^c and $t_{i,k} < t_{i,m}$. We need to show that $t_{i,k} < t_{i,m}$ in $z(n)$. There are two cases. First, suppose that $t_{i,k}$ and $t_{i,m}$ are in the same commit segment σ. Since $\sigma \sqsubseteq z(n)$ and the commit segment σ is the superset that contains the transaction ADS segments in which $t_{i,k} < t_{i,m}$, it follows that $t_{i,k} < t_{i,m}$ in $z(n)$. Second, suppose that $t_{i,k}$ is in commit segment σ_x while $t_{i,m}$ is in σ_y and $\sigma_x < \sigma_y$. Since $\sigma_x < \sigma_y$ in z_n, it follows that $t_{i,k} < t_{i,m}$ in $z(n)$.

We now prove that all the transaction ADS segments in an elementary transaction are interleaved serializably with those of others in $z(n)$. First, it follows from Assumption 6.2 that steps of elementary transactions are interleaved setwise serializably in z and in the remaining partial schedules z_i, $1 \leq i \leq n$. Since $z^c \sqsubseteq z$ and $z_i^c \sqsubseteq z_i^c \sqsubseteq z_i$, $1 \leq i \leq n$, it follows that transaction ADS segments are interleaved serializably with those of others in each of the committed partial schedules z_i^c, $1 \leq i \leq n$. We now claim that all the transaction ADS segments in an elementary transaction are interleaved serializably with those of others in $z(n)$. If we suppose that this claim is false, then there must exist at least two transaction ADS segments in an elementary transaction such that these two segments are interleaved non-serializably. Let these two ADS segments be σ_1 and σ_2. There are two cases. First, σ_1 and σ_2 are in the same committed partial schedule. This contradicts the result that all the ADS segments of an elementary transaction are interleaved serializably with those of others in the same committed partial schedule. Second, suppose that σ_1 and σ_2 are in two different committed partial schedules z_1^c and z_2^c and that $z_1^c < z_2^c$. By the assumption that these two transaction ADS segments are non-serializable, there must exist some steps $t_{2,j} > t_{2,k}$ in σ_2 and some steps $t_{1,m} > t_{1,n}$ in σ_1 such that "$t_{1,j} < t_{2,m}$" and "$t_{1,k} > t_{2,n}$". However, this contradicts the fact that all the steps in committed partial schedule z_1^c precede those in z_2^c. Hence, all the transaction ADS segments of an elementary transaction are interleaved serializably with those of others in $z(n)$. □

Corollary 6.5: The failure safe rule is consistent and correct.

Theorem 6.6: The failure safe rule $\mathfrak{R}$ is the optimal failure recovery rule within the set of all the modular and safe failure recovery rules for transaction systems using generalized setwise serializable scheduling rules.

Proof: To prove the optimality of the failure safe rule $\mathfrak{R}$, we need to show that the commit segments produced by $\mathfrak{R}$ for any transaction is the most refined partition. There are two cases. First, a transaction ADS segment of transaction T_i could be taken as a commit segment by $\mathfrak{R}$. By Theorem 6.3, this commit segment cannot be further partitioned. In the second case, transaction ADS segments $\sigma_1, \ldots, \sigma_m$ in an elementary transaction of T_i are taken as a single commit segment by $\mathfrak{R}$, because they are interleaved. Suppose that they are interleaved and taken as a single atomic commit segment σ. Suppose that there exists a modular and safe recovery rule $\mathfrak{R}^*$ which divides σ into more than one atomic commit segments, $\sigma_{c,1}, \ldots, \sigma_{c,n}$, where $n \geq 2$. By Theorem 6.3, each of these commit segments must contain an integer number of transaction ADS segments. Therefore, $\sigma_{c,1}, \ldots, \sigma_{c,n}$ must be supersets of transaction ADS segments. Let the first failure occur just after the commit of $\sigma_{c,1}$ and let there be no more failures. Suppose that the ordering of steps of T_i in $z(1)$ is consistent with the internal ordering of steps in T_i. This contradicts the assumption that the steps of the transaction ADS segments in σ are interleaved, because in $z(1)$ all the steps of $\sigma_{c,1}$ in z^c precede all the steps in $\sigma_{c,2}, \ldots, \sigma_{c,n}$ in z_1. It follows that steps of T_i in $z(1)$ violates the internal ordering of steps in T_i and therefore $z(1)$ is not generalized setwise serializable. This contradicts the assumption that $\mathfrak{R}^*$ is safe. Thus, the commit segments produced by $\mathfrak{R}$ for any transaction T_i cannot be refined by other modular and safe recovery rules. □

6.3 Summary

In this chapter, we have shown that when the scheduling rule is the generalized setwise serializable scheduling rule, the optimal modular and safe recovery rule is the *failure safe rule*. Given a compound transaction, the failure safe rule examines the transaction ADS segments in each of the elementary transactions. If an ADS segment is not interleaved with other transaction ADS segments in the same elementary transaction, then this ADS segment is an atomic commit segment. Interleaved ADS segments in an elementary transaction must be merged to form a single atomic commit segment.

From an application point of view, there are two important points one should be aware of. First, one

should write one's compound transaction carefully to avoid the interleaving of transaction ADS segments, whenever this is possible. Unnecessarily interleaving the steps of transaction ADS segments in an elementary transaction could lead to a serious loss of system concurrency. Second, one must realize that the spirit of the failure safe rule is to provide "check-points", so that a transaction can resume its execution after being interrupted by failures. As a matter of fact, once we have committed one single atomic commit segment of a transaction, we can only abort commit segments not yet committed. We cannot this transaction as a whole. Once we commit a commit segment we are obliged to complete the execution of our transaction. For example, in the compound transaction Get-A-and-B introduced in the overview of this thesis, once we have obtained one unit of a resource and committed the operation, we are obliged either to get the other unit or to put back the unit which we have already taken.

We give a few informal suggestions regarding the implementation of the failure safe rule. The standard distributed version of the two phase commit protocol[3] [Bernstein 83] can be adopted to commit those atomic commit segments. The major modification is that we also need to store the values of local variables of a commit segment in the stable storage. Saving these values in the stable storage is important for resuming the execution of a transaction that has been interrupted by a failure. To optimize the storage utilization, we can save only those local variables that are shared by different commit segments. The identification of those shared variables can be made easy if one is willing to request the explicit declaration of them as *atomic variables* in the program of a compound transaction.

Finally, we want to comment on the orphan problem of a nested transaction, because this is a frequently discussed topic [Goree 83]. This problem occurs when a higher level nested transaction aborts but the lower level ones still running. For example, supposed that nested transaction T is running at the home node and invokes a sub-transaction at a remote node. If transaction T is aborted by a failure at the home node then the sub-transaction running at the remote node becomes an orphan. It is important to point out that this orphan problem is an optimization issue, not a database consistency or transaction correctness issue. This is because a sub-transaction of a nested transaction cannot commit its results to the database. When an orphan sub-transaction eventually finishes its execution, it will try to pass its lock and results to its parent. If its parent cannot be contacted by the database system, then the sub-transaction will be aborted by the recovery manager of the database system. Since an orphan is destined to be aborted and it consumes resources, it is desirable to track down orphans and abort them as quickly as possible.

---

[3] Some calls it three phase commit protocol.

Since our compound transactions are partitioned into a partially ordered set of commit segments by our failure safe rule and each of the commit segments is committed in an order that is consistent with the partial order of commit segments, we do not have the orphan problem at the level of managing commit segments of a compound transaction. However, within a commit segment we can have the orphan problem because a commit segment can be an elementary transaction in a nested form. The investigation of the orphan problem is beyond the scope of this thesis and we suggest the use of algorithms in the database literature.

7. An Example of Application

The two main aspects in the application of our theory are to partition the database into consistency preserving atomic data sets and to write and schedule compound transactions. In this chapter, we give an example to illustrate these two aspects. In Section 7.1, we give a brief review of our theory from an application point of view. In Section 7.2, we address the partition of the database and in Section 7.3 we discuss the writing of compound transactions. Readers who are interested in more detailed examples in the context of distributed operating systems and readers who are interested in the integration of our concurrency control and failure recovery rules with the object oriented (abstract data type) approach, are recommanded to read Tokuda's, Clark's and Locke's work [Tokuda 85]. One of their main research efforts is to develop a new computational model called Arobjects which integrates our theory with distributed abstract data types. A brief review of this effort is in Section 8.2, future work.

7.1 A Brief Review of Our Theory

Conceptually, while atomic data sets result from a consistency preserving partition of the database, elementary transactions result from a correctness preserving partition of a transaction. Given a database and its associated consistency constraints, we partition the database into consistency preserving atomic data sets. As long as the consistency of each atomic data set is satisfied individually, the consistency of the database is also satisfied. Given a transaction and its associated post-conditions, we partition the transaction into a partially ordered set of correctness preserving elementary transactions. As long as the post-condition of each elementary transaction is individually satisfied, the post-conditions of the compound transaction are also satisfied.

Each elementary transaction must preserve the consistency of the database and satisfy its own post-condition when it is executed serially and in an order consistent with the partial ordering of elementary transactions in the compound transaction. Given a compound transaction, the generalized setwise serializable scheduling rule partitions the steps of each elementary transaction into transaction ADS segments. A transaction ADS segment is just all the steps in an elementary transaction that access the same atomic data set. This means that as long as we execute all the elementary transactions setwise serializably and in a order consistent with the partial orderings of elementary transactions defined by compound transactions, the concurrency execution of compound transactions will be consistent and correct.

To cope with system failures, the failure safe rule examines the transaction ADS segment of each elementary transaction. If the steps of a transaction ADS segment are not interleaved with the steps of other transaction ADS segments, then this segment is named as a commit segment. If the steps of several transaction ADS segments are interleaved with each other, then they are merged into a single commit segment. This procedure creates a partially ordered set of commit segments. As long as each compound transaction commits its commit segment in an order consistent with the partial ordering of its commit segments, the concurrent control will remain consistent and correct despite clean and soft system failures.

In order to avoid cascaded aborts the writelocks acquired by a commit segment cannot be released until it has successfully committed. Thus, for all practical purpose commit segments are the atomic units for both concurrency control and failure recovery in a compound transaction. As a style of programming, we prefer to make elementary transactions correspond to commit segments: our atomic units for both concurrency control and failure recovery. That is, we would like to write compound transactions in such a way that each elementary transaction contains a single commit segment. This tends to simplify the implementation of the concurrency control and failure recovery of compound transactions, because we can use the scope of elementary transaction as delimiters for the runtime management of concurrency control and failure recovery.

7.2 Partitioning The Database

From an application point of view, to partition the database into consistency preserving atomic data sets is straightforward if the consistency constraints are given. However, the specification of database consistency constraints needs a good understanding of the applications in question.

In order to maximize the benefit of parallelism in a distributed system, it is often useful to consider the use of consistency constraints that are weaker than the corresponding ones in centralized systems. We illustrate this idea by a simplified case of managing the directories of a file system as follows. We consider a set of shared system files distributed at different nodes. There is a local directory (LD) at each node indicating the resident files. With only these local directories, one must potentially search through all the LD's in order to locate a file, and this would be very inefficient. To increase efficiency, the system has a global directory (GD) at some node. The GD indicates which LD should be searched for each of the shared files. The GD is replicated for reliability and performance. When one needs a file, the local operating system kernel will first search through its LD, and then it will search a nearby GD, if the file is not in its LD. The introduction of

Corollary 6.5: The failure safe rule is consistent and correct.

Theorem 6.6: The failure safe rule $\mathfrak{R}$ is the optimal failure recovery rule within the set of all the modular and safe failure recovery rules for transaction systems using generalized setwise serializable scheduling rules.

Proof: To prove the optimality of the failure safe rule $\mathfrak{R}$, we need to show that the commit segments produced by $\mathfrak{R}$ for any transaction is the most refined partition. There are two cases. First, a transaction ADS segment of transaction T_i could be taken as a commit segment by $\mathfrak{R}$. By Theorem 6.3, this commit segment cannot be further partitioned. In the second case, transaction ADS segments σ_1, ..., σ_m in an elementary transaction of T_i are taken as a single commit segment by $\mathfrak{R}$, because they are interleaved. Suppose that they are interleaved and taken as a single atomic commit segment σ. Suppose that there exists a modular and safe recovery rule $\mathfrak{R}^*$ which divides σ into more than one atomic commit segments, $\sigma_{c.1}$, ..., $\sigma_{c.n}$, where $n \geq 2$. By Theorem 6.3, each of these commit segments must contain an integer number of transaction ADS segments. Therefore, $\sigma_{c.1}$, ..., $\sigma_{c.n}$ must be supersets of transaction ADS segments. Let the first failure occur just after the commit of $\sigma_{c.1}$ and let there be no more failures. Suppose that the ordering of steps of T_i in $z(1)$ is consistent with the internal ordering of steps in T_i. This contradicts the assumption that the steps of the transaction ADS segments in σ are interleaved, because in $z(1)$ all the steps of $\sigma_{c.1}$ in z^c precede all the steps in $\sigma_{c.2}$, ..., $\sigma_{c.n}$ in z_1. It follows that steps of T_i in $z(1)$ violates the internal ordering of steps in T_i and therefore $z(1)$ is not generalized setwise serializable. This contradicts the assumption that $\mathfrak{R}^*$ is safe. Thus, the commit segments produced by $\mathfrak{R}$ for any transaction T_i cannot be refined by other modular and safe recovery rules. □

6.3 Summary

In this chapter, we have shown that when the scheduling rule is the generalized setwise serializable scheduling rule, the optimal modular and safe recovery rule is the *failure safe rule*. Given a compound transaction, the failure safe rule examines the transaction ADS segments in each of the elementary transactions. If an ADS segment is not interleaved with other transaction ADS segments in the same elementary transaction, then this ADS segment is an atomic commit segment. Interleaved ADS segments in an elementary transaction must be merged to form a single atomic commit segment.

From an application point of view, there are two important points one should be aware of. First, one

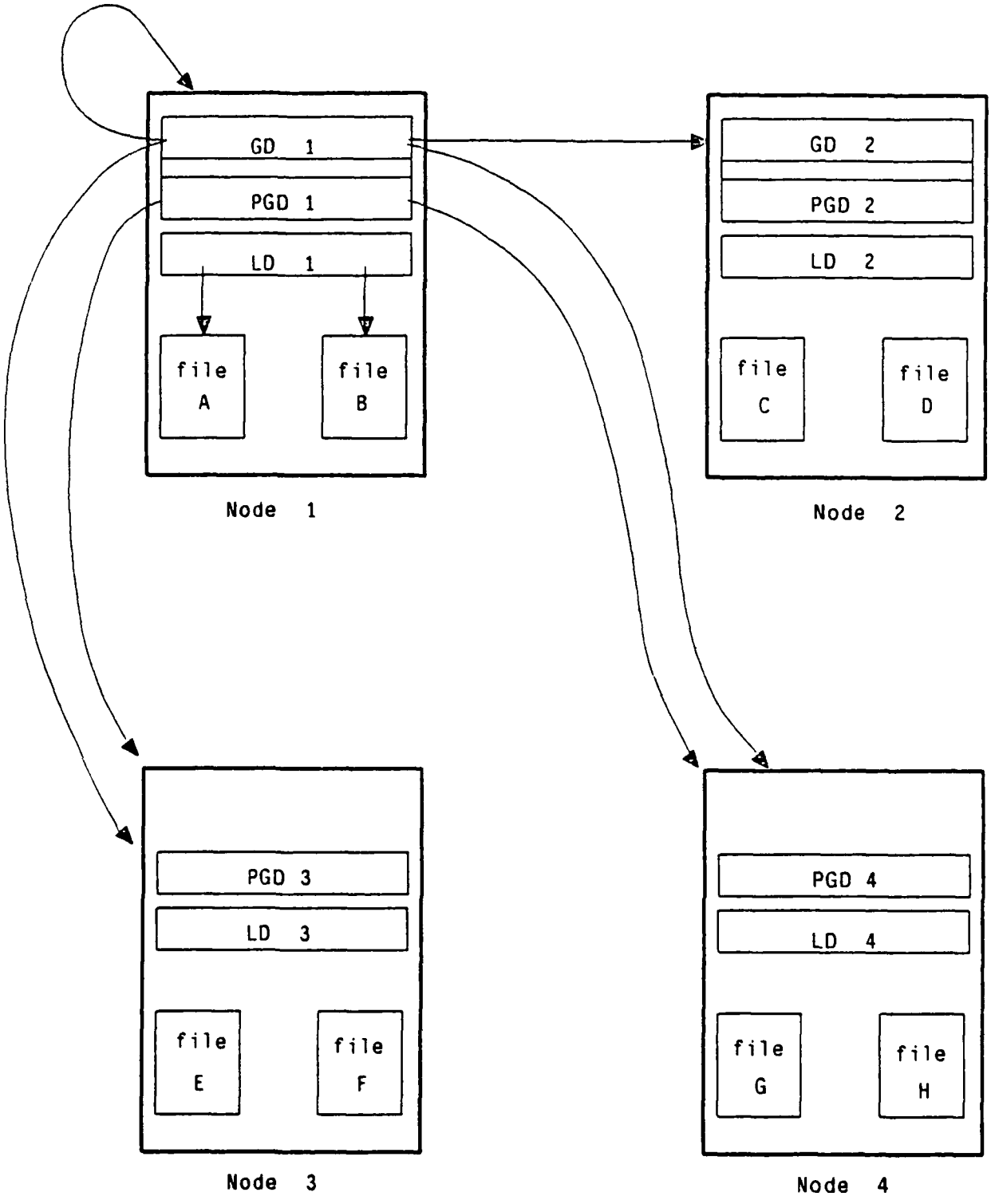


Figure 7-1: A Distributed Directory System

GD but do not need to use concurrency control protocols to make the transfer of the file and the updating of the LD and GD appear to be an instantaneous event with respect to other transactions. On the other hand, if we want to enforce the consistent constraint "each of the file entries in a GD always points to the current location of the file", then a transaction must use concurrency control protocols to ensure that the transfer of the file and the updating of the GD and the LD appears to be an instantaneous event.

From an application point of view, although all the consistent states are equivalent in the sense that none of them will cause transactions to execute incorrectly, certain consistent states are more favorable than others. Performance enhancement schemes are designed to increase the *probability* of the database staying in the favorable states. For example, suppose that we replace the strong consistency constraint "each of the file entries in a GD always points to the current location of the file" with the weak one "each of the file entries of a GD can point to either the current location or a historical location of the file". In this case, the states in which file entries of a GD point to the old locations are consistent but unfavorable states. The performance enhancement scheme "update the GD whenever the location of a file changes" will only improve the chance that transactions will see the consistent and favorable state "each of the file entries in a GD points to the current location of the file." However, this is not a guarantee. It is possible that when transaction T_1 transfers file F from node 1 to node 3 and updates the LD at node 1, another transaction T_2 is reading the GD at node 2, which indicates that file F is in node 1. When transaction T_1 is updating the GD at node 2, T_2 searches the LD at node 1 to locate file F and only finds that file F is no longer in node 1.

Generally speaking, weaker consistency constraints permit a higher degree of concurrency. However, once the consistency constraints are weakened, the complexity of transactions will be increased for two reasons. First, the process of weakening the consistency constraints enlarges the number of system states that are considered to be consistent. For example, if the set of consistency constraints regarding the GD and PGD is relaxed, a transaction must be written to function correctly in the case that the GD or PGD will only give a *valid* LD location but not necessarily the LD location where the file actually resides. That is, a transaction must be able to abort when the file cannot be found or traced. Transactions must have the ability to deal with all the possible system states that are consistent. Second, strong consistency constraints generally ensure that the system will stay in a small set of favorable states, although enforcement could be too expensive. When the set of permissible states is enlarged, the post-conditions of transactions must be redesigned to better keep the system in favorable states. This also increases the complexity of transactions. The evaluation of the trade-offs between system concurrency and transaction complexity is an exciting new research area. However, the

analysis of the performance trade-off is outside the the scope of this thesis. As a rule of thumb, however, it is usually worthwhile to have weaker consistency constraints if the resulting increase of complexity in transactions is manageable, such as it is in the example.

7.3 Writing Compound Transactions

From a programmer's point of view, under serializability theory the entire set of steps of a transaction is a single unit of atomic action, that is, it is a single commit segment. Under our theory, a compound transaction is a partially ordered set of atomic actions in the form of atomic commit segments. When an atomic action is committed, it leaves the database consistent. Therefore, the results of its computation can be seen by other transactions and the locks held by it can be released.

In order to improve system concurrency, we should try to have many small atomic actions rather than a few big ones in the writing of a compound transaction. Since the consistency of each atomic data set can be preserved independently of others, one way to have small atomic actions is to try grouping the operations on each of the atomic data sets into elementary transactions. There are three guiding principles in doing so.

1. Each of these elementary transactions must preserve the consistency of the accessed atomic data sets so that other transactions will not see inconsistent states.

2. The compound transaction must have a method to handle the consequence of releasing the locks by its elementary transactions. This is because when the locks on an atomic data set are released, the state of this atomic data set is likely to be changed by other transactions.

3. Committing an elementary transaction is as serious as committing an entire transaction under a serializable schedule. We cannot abort a committed action: an entire transaction or an elementary transaction. Once we have committed an elementary transaction of a compound transaction, we are obliged to complete the execution of the compound transaction. For example, in the compound transaction Get-A-and-B discussed in the overview, if we have obtained one of the resources and committed the elementary transaction, then we are obliged to complete the execution of Get-A-and-B by either putting back the obtained resource or getting another one.

We now illustrate the construction of a compound transaction by a file transfer transaction example.

Suppose that in the directory example presented in the previous section we have two global directories: GD1 and GD2, a set of local directories: LD1, ..., LDn and a set of partial global directories: PGD1, ..., PGDn. According to the discussion in the previous section, each of them is modelled as an atomic set. In this

example, we assume that each PGD and each GD is a list of <file name, name of resident node> pairs. However, each LD directory has a tree structure. For simplicity, we assume that each tree has only two levels: the root and the leaves. The root is a list of <file name, file descriptor pointer> pairs. Each of the leaves represents a file descriptor which is a record of various attributes such as logical name, system name, the storage area, the capability attribute and the file state attribute. The capability attribute describes the current state of user capabilities such as, it is a read only file for all users except for the one who holds a special key. The file state attribute describes who holds a readlock or writelock on the file. For simplicity, we also assume that all the system files have unique logical names given by some system name server. Figure 7-2 shows the tree structure of a local directory.

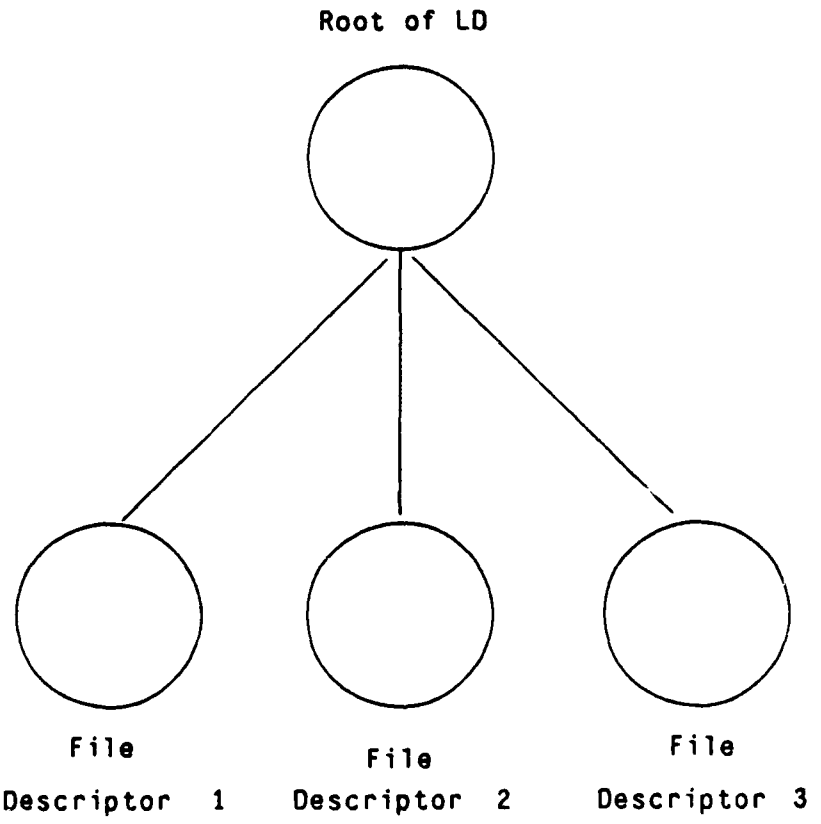


Figure 7-2: A Local Directory

The tree structure of a LD directory suggests using the tree lock protocol [Silberschatz 80]. To enforce

setwise serializability in the execution of elementary transactions, we must observe the following protocol on each of the tree structured atomic data sets:

- except for the first data object locked, no data object in a tree can be locked unless a lock is currently held on its parent.

- no data object is ever locked twice, that is, an data object cannot be unlocked and then locked again later by the same elementary transaction.

To avoid cascaded aborts, we have the additional requirement that each elementary transaction keeps all the writelocks until it has committed. For unstructured atomic data sets we will use the setwise two phase lock protocol. This protocol requires an elementary transaction not to release any lock until it has acquired all the locks in an atomic data set. To avoid cascaded aborts, we will keep all the writelocks acquired in an atomic commit segment until the atomic commit segment has successfully committed. Note that an atomic commit segment can include several interleaved transaction ADS segments. This means that as long as readlocks are concerned, we can release them by following the setwise two phase lock protocol. But the release of writelocks will be further delayed until the atomic commit segment has successfully committed.

A file transfer transaction will be given the file name, the capability key, the names of the source node and the destination node. If this transaction cannot find the file in the source node, it will report to the user that his file is not in the source node and quit. In addition, if the transaction finds out that the user's capability key does not allow him to move this particular file, he will also be informed accordingly. Otherwise, it will transfer the file from source node to the destination node, leave a forwarding address at the PGD of the source node and update the source LD, destination I D, GD1 and GD2.

Before designing the compound transaction, we would like first to give a more detailed description of this transaction. The first atomic data set accessed by this compound transaction is the source I.D. We begin with an elementary transaction Check-Out which first readlocks the root of source LD and checks if the file is in this node. If it cannot find the file name, it will report to the user that his file is not in the source node and then quit. If it finds the file, then it writelocks the descriptor of this file and checks if the user has the capability to move this file. If the user does not have the required capability, he will be informed accordingly and our compound transaction stops here. If the user has the capability to do so, then Check-Out first copies the file descriptor to a private area in the stable storage and then updates the capability attribute of this file descriptor to disable the capabilities of write, delete and transfer for all other transactions. However, users

who have the capability to read this file can still read it. After we have copied the file from the source node to the destination node and updated the source PGD, GD1 and GD2, we will wait for those read transactions finishing their reading and then delete the source file and update the source LD. This arrangement will provide an un-interrupted service to users of the system files.

Having committed elementary transaction Check-Out, we start an elementary transaction Copy-File-to-Buffer, which copies both the saved original file descriptor and the file into a buffer in the source node.[4] Having copied the file into the source buffer, we start another elementary transaction Send, which sends the content of the source buffer to the buffer at the destination node. After sending the file and its descriptor to a buffer in the destination node, we start a new elementary transaction Copy-File-from-Buffer which writelocks the root of the destination LD, copies the file from the buffer to the disk, updates the list of <file name, file descriptor pointer> pairs at the root and creates the corresponding file descriptor at the destination node for the transferred file. After the file is copied over, we start three elementary transactions in parallel. One is to leave a forwarding address in the PGD of the source node, one is to update GD1 and the last one is to update GD2. After the source PGD, GD1 and GD2 are updated, we are now ready to delete the file at the source node. We start a new elementary transaction Delete to delete the file, file descriptor and the corresponding <file name, file descriptor pointer> at the root of the source LD.

In the following, we first outline the structure of compound transaction Transfer. Next, we comment on the implementation of recovery management. Finally, we give the pseudo-code of compound transaction Transfer.

```
CompoundTransaction Transfer
AtomicVariable continue : Boolean;
    SourceBuffer, DestinationBuffer: array[1..size] of char;
BeginSerial
  ElementaryTransaction Check-Out;
  {This transaction sets the variable continue to
   either true or false according to user's capability.}
```

[4] For simplicity, we assume that we have large buffers each of which can hold an entire file.

```
if not continue then goto Last-Step

ElementaryTransaction Copy-File-to-Buffer;

ElementaryTransaction Send;

ElementaryTransaction Copy-File-from-Buffer;

BeginParallel
  ElementaryTransaction Update-PGD;

  ElementaryTransaction Update-GD1;

  ElementaryTransaction Update-GD2;
EndParallel;

ElementaryTransaction Delete;

Last-Step: Commit and release all the locks;
EndSerial. {end of compound transaction Transfer}
```

Having outlined the structure of compound transaction Transfer, we now comment on the recovery management of this transaction.

```
CompoundTransaction Transfer
AtomicVariable continue : Boolean;
    SourceBuffer, DestinationBuffer: array[1..size] of char;
BeginSerial
  ElementaryTransaction Check-Out
  {Checks the user capability. If o.k. then updates the capability
   attribute of the file descriptor and sets continue to true.}

  {When Check-Out commits, the value of continue is saved at disk.
   In addition, the recovery manager check-points that the first
   commit segment of Transfer has been committed. If a failure occurs
   after the commit of Check-Out and before the second commit, the
   execution of Transfer will be re-started at the first step after
   Check-Out and the values of atomic variables are restored from disk.
   We must emphasize that the commit of the results of an elementary
   transaction, the save of the values of atomic variables and the
   check-point of the execution of the compound transaction must
   be done atomically: either all three is done or none is done.}
```

```
If not continue then goto Last-Step;

ElementaryTransaction Copy-File-to-Buffer
{Copies the file and its descriptor to the source node  buffer.}
{Commit point and execution check-point.}

ElementaryTransaction Send
{Sends the content of the source buffer to the destination buffer.}
{Commit point and execution check-point.}

ElementaryTransaction Copy-File-from-Buffer
{Copies the file from destination buffer to disk and updates the LD.}
{Commit point and execution check-point.}

BeginParallel
  ElementaryTransaction Update-PGD;
  {commit point and execution check-point.}

  ElementaryTransaction Update-GD1;
  {commit point and execution check-point.}

  ElementaryTransaction Update-GD2;
  {commit point and execution check-point.}

EndParallel;

{These three elementary transactions will be executed, committed and
 check-pointed in parallel. For example, if only one is successfully
 committed and check-pointed but the other two are aborted by a
 failure, then the aborted two will be restarted in parallel.}

ElementaryTransaction Delete;
{Deletes the file at source node and updates LD.}
{Commit point and execution check-point}

Last-Step: Commit and release all the locks;
EndSerial.{end of compound transaction Transfer}
```

Having discussed the structure of compound transaction Transfer, we now list the pseudo-code of this transaction.

```
CompoundTransaction Transfer(file-name, capability-key,
                                        source-node, destination-node);
AtomicVariable continue : Boolean;
    SourceBuffer, DestinationBuffer: array[1..size] of char;
BeginSerial
  ElementaryTransaction Check-Out(file-name, capability-key);
  BeginSerial
    Readlock the root of source LD;
    If file is not here
    then BeginSerial
           Report "file  is not in source node";
           continue := false;
         EndSerial
    else continue := true;

    if continue
    then BeginSerial
           Writelock the file descriptor;
           Check if the user has the capability to move the file;
           if he has the capability
           then BeginSerial
                  Copy the descriptor to a private area in disk;
                  Update the capability attribute to disable the delete,
                  write and transfer capabilities of other transactions.
                EndSerial
           else BeginSerial
                  continue := false;
                  Report to user that he cannot move the file;
                EndSerial;
         EndSerial;
    Commit and release all the locks;
    {The atomic variable continue is saved at stable
     storage as part of the commit operation.}
  EndSerial;{end of  Check-Out}


  If not continue then goto Last-Step;
```

```
ElementaryTransaction Copy-File-to-Buffer(file-name, buffer-name);
BeginSerial
  Readlock the root of source LD;
  Readlock the file descriptor;
  Copy the file descriptor
  saved in a private area of the stable storage into the buffer;
  Readlock the file;
  Copy the  file into the buffer;
  Commit and release all the locks;
EndSerial;{end of Copy-File-to-Buffer}

ElementaryTransaction Send(source-buffer-name,
                           destination-buffer-name);
BeginSerial;
  Copy the content of the source buffer to the destination buffer;
  Commit;
EndSerial;{end of Send}

{Buffers are private storage area, no lock is needed.}

ElementaryTransaction Copy-File-from-Buffer(file-name,
                                            destination-buffer-name);
BeginSerial
  Writelock the root of destination LD;
  Copy the file from the buffer to disk;
  Create a new file descriptor;
  Copy all the attributes from the original one
  except storage area and file state attributes;
  In the storage area attribute, record the new storage area;
  In the file state attribute,
  record that currently there is no lock on the file;
  Insert  <file name, pointer to file descriptor> pair to the root;
  Commit and release all the locks;
EndSerial;{end of Copy-File-from-Buffer}



BeginParallel
  ElementaryTransaction Update-PGD(file-name, source-node);
  BeginSerial
    Writelock the  source PGD;
    Leave  the new address of the file in PGD;
    Commit and release all the locks;
  EndSerial;{end of Update-PGD}
```

```
    ElementaryTransaction Update-GD(file-name, GD1);
    BeginSerial
      Writelock GD1;
      Update the transferred file's address;
      Commit and release all the locks;
    EndSerial;{end of Update-GD}


    ElementaryTransaction Update-GD(file-name, GD2);
    BeginSerial
      Writelock GD2 file name entry;
      Update the transferred file's address;
      Commit and release all the locks;
    EndSerial;{end of Update-GD}
  EndParallel;

  ElementaryTransaction Delete(file-name, source-node);
  BeginSerial
    Writelock the root of source LD;
    Writelock the file descriptor;
    Writelock the file;
    Delete the file;
    Delete the file descriptor;
    Delete the <name, file descriptor pointer> pair at the root;
    Commit and release all the locks;
  EndSerial;{end of Delete}
  Last-Step: Commit and release all the locks;
  {In this compound transaction, this step is only a formalism in
   the syntax. All the locks have been released by the elementary
   transactions.}
EndSerial.{end of Transfer}
```

Since we divide compound transaction Transfer into many elementary transactions each of which commits and releases its acquired locks, we obtain a very good degree of concurrency. We want, however, to make two comments on the first elementary transaction Check-Out. First, the manipulation of the capability attribute by Check-Out is equivalent to having atomic readlocks at the file and at the file descriptor. An atomic readlock is a readlock recorded in the stable storage and therefore will not be affected by failures. Instead of manipulating the capability attribute, one can use atomic readlocks if one's system supports them. Although syntactically, each of our elementary transactions, including Check-Out, commits and releases all its acquired locks, Check-Out, in effect, passes the "atomic readlocks" to compound transaction Transfer rather than releasing all the locks after it has committed.

We have mentioned that when an elementary transaction commits, it *could* release all the locks without

adversely affecting other transactions because it leaves the database in a *consistent* state. However, releasing all the locks will allow other transactions to modify the atomic data sets in an unrestricted way such as deleting the file, and our compound transaction must devise a way to cope with it. Depending upon the application in question, sometimes it is easy to devise such a way and sometimes it is not. In this particular instance, releasing all the locks provides a high degree of concurrency but also makes the compound transaction Transfer very complex.

The easiest way to reduce the complexity of compound transaction Transfer is to extend the scope of Check-Out. That is, we make the first elementary transaction to be Check-Out-and-Transfer. Since we now hold the writelocks on both the source LD and destination LD within a single elementary transaction , the problem mentioned above cannot happen. Although this solution results in a simple compound transaction, the concurrency is low, because the transaction holds writelocks on both source and destination LD's while transferring a file across a network.

A reasonable trade-off between the system concurrency and transaction complexity is to let Check-Out modify the capability attribute or use the atomic readlocks as described before. Generally speaking, having committed an elementary transaction we have two options. One is to release all the locks and deal with the consequence later. Another is to use atomic readlocks so that the resulting computation is visible to but not changeable by other transactions. This is a compromise between the approaches of "ending the elementary transaction and releasing all the locks" and of "extending the scope of the elementary transaction and holding on all the locks". In this instance, we consider this compromise approach is a good choice for our problem at hand. Recall that in the previous section, we have said that the trade-offs between system concurrency and transaction complexity is an important consideration in the design of atomic data sets. We now see that it is also an important consideration in the design of a compound transaction.

8. Conclusion

8.1 Summary

We have developed a formal theory of modular scheduling and recovery rules. This theory coherently addresses the notions of consistency, correctness, modularity and optimality in concurrency control and failure recovery. Furthermore, this theory provides us with provably consistent, correct and optimal modular concurrency control and failure recovery rules.

The four main results in this theory are a formal model of modular scheduling rules, the setwise serializable scheduling rule, the compound transactions and their associated generalized setwise serializable scheduling rules, and the failure safe rule. The formal model of modular scheduling rules permits us to have a unified treatment of different modular scheduling methods. The setwise serializable scheduling rule is the optimal application independent scheduling rule. The generalized setwise serializable scheduling rules form a complete class within the set of all the modular scheduling rules. This means that for any given modular scheduling rule, there always exists a generalized setwise serializable scheduling rule that provides at least as much concurrency. The failure safe rule is the optimal failure recovery rule for any given set of compound transactions scheduled by the generalized setwise serializable scheduling rules. Finally, our theory includes nested transactions, serializability theory and failure atomicity as special cases.

8.2 Future Work

Our work on modular concurrency control and failure recovery rules only addresses one of the many inter-related topics in the study of transaction facilities of a distributed computer system. There are many related topics that need to be investigated. For example, the performance evaluation of the trade-offs between system concurrency and transaction complexity mentioned in Chapter 7 is a new area of research. We believe that the proper specification of consistency constraints of a distributed computer system is of fundamental importance. When the performance of a distributed computer system becomes a problem, many practioners will use some form of ad hoc "weak" consistency constraints to allow for a higher degree of system concurrency. It would be helpful to have a formal theory to guide the trade-offs between system concurrency and transaction complexity.

Another interesting area of research is the concept of *co-operating transactions* [Sha 83]. Co-operating

transactions are a model of a team of co-operating transactions which co-ordinate their actions without a common top level transaction. As an analogy, co-operating transactions are like co-routines rather than sub-routines of a higher level program. Currently, concurrency control theories do not allow transactions to exchange messages and co-operate. From a computational point of view, these transactions are merely processes with embedded concurrency control and failure recovery mechanisms. Co-operating (communicating) processes play an important role in modern operating systems, but unfortunately we do not have a theory of co-operating transactions. In a previous paper [Sha 83], we reported some experiments with the idea of co-operating transactions: a set of transactions with a set of consistency constraints defined upon their state variables. These consistency constraints represent the rule of co-operation. However, the formalization of these ideas still has a long way to go.

The direct extentions of our work are those that are directly related to the development and implementation of transaction facilities. To implement a transaction facility that supports the application of our theory, we need to develop programming languages and their compiler/interpreter that support the writing of compound transactions, a distributed database system or a distributed operating system that provides runtime support for the implementation of atomic data sets and the execution of compound transactions. To use our theory for the construction of an operating system rather than database applications, we need to integrate our concurrency control and failure recovery rules with the software engineering approach: abstract data types.

Although this future work is beyond the scope of this thesis, we are glad to report that progress has already been made. Currently, in the Computer Science Department of Carnegie-Mellon University, Tokuda, Clark and Locke are developing a decentralized operating system ArchOS [Tokuda 85], which is the operating system of the Archons decentralized computer system project [Jensen 83, Jensen 84]. An important feature of ArchOS is that it supports transactions at the kernel level. In fact, all the ArchOS system primitives employ transactions. Their research effort in the area of transaction facility includes,

- The development of a new model of computation called Arobject. This model integrates our theory with the distributed abstract data type approach.
- The formulation of an alternative syntax of compound transactions, which is more suitable for the implementation of operating system objects.
- The development of various mechanisms that carry out our concurrency control and failure recovery rules.

- The design and implementation of a prototype system.

Their computation model integrates our theory with distributed abstract data type approach. In this thesis, a database is modelled as a set of shared data objects, which is not constructed as *layered* and *encapsulated* objects in the form of abstract data types. To apply our theory to the construction of operating system objects, we need to address the problem of integrating existing lower level operating system objects into a higher level one. Although currently Arobject is still in a developmental stage, we would like to describe some of the basic ideas for solving the abstraction problem. These ideas are important for the application of our theory in the context of operating systems.

In essence, the idea is to divide an Arobject into well defined levels (layers) of abstractions and then apply our theory to each of the levels for concurrency control and failure recovery management. An Arobject can consist of distributed data objects and their associated operations with many levels of abstraction. At level zero of abstraction, the state space of an Arobject is the Cartesian products of the domains of data objects at level zero. An Arobject designer chooses a set of consistency constraints that defines a subset of the state space called the consistent states at level zero. A set of consistent and correct transactions is then written to represent the operations of at level zero. Our theory is then used for the concurrency control and failure recovery of this set of transactions.

Having developed the level zero Arobject, the set of consistent states at level zero is then partitioned into equivalent classes each of which corresponds to an abstract state visible at level one. Viewed at level one, each of the transactions that can be called at level one represents a non-divisible primitive step (operation) at level one, which transforms a level one state to another level one state. At level one, an Arobject designer can define new consistency constraints upon the abstract state space at level one and write transactions composed by the primitive operations provided by level zero. A level one designer needs not worry about how to maintain the consistency constraints at level zero. These constraints are automatically maintained by the primitive operations at level one. Our theory can then be applied again at level one for the purpose of concurrency control and failure recovery of transactions written at level one. This exercise can be repeated again and again to produce higher levels of abstraction.

As an example of this approach, suppose that Disk-1 is an Arobject at level zero. At this level, Disk-1 controls the disk driver, formats the disk and keeps track of the physical addresses of each block and and its

state: empty or allocated to user i. Supposed that there is a total of 100,000 blocks of storage at disk one. An abstract state provided by level zero to level one is in the form of <number-of-empty-blocks, number-of-blocks-used-by-user-i, $0 \leq i \leq$ maximal-number-of-users> with associated consistency constraint "the sum of empty blocks and allocated blocks equals to 100,000". A level one state describes the number of empty blocks and how many blocks used by each of the users, but ignores the detailed information such as the physical locations of user i's blocks. The primitive operations provided by level zero to level one are transactions Allocate-Blocks and Deallocate-Blocks. Note that the consistency constraints at level zero such as those associated with storage locations (map) and formating are not visible at level one but will be enforced by transactions Allocate-Blocks and Deallocate-Blocks. As long as a level one designer is concerned, these two transactions play the role of primitive atomic steps at his level. Each of this two steps maps a level one state to another level one state. A level one state describes the numbers of empty blocks and the numbers of blocks allocated to each of the users. The fact that these two steps are actually complex transactions which controls the disk driver and maintains many level zero consistency constraints such as keeping track of physical locations should not be a concern of a level one designer.

Let us develop the example one step further. Suppose that we have ten such Arobjects Disk-1, ..., Disk-10. In this case, the abstract state space at level one is the Cartesian products of the abstract state space provided by each of the disks. That is, the state space at level one is all the possible combinations of allocating (10 x 100,000) blocks to users. At level one, Arobject Disk-System can define new consistency constraints upon the abstract state space provided by level zero. For example, in addition to provide 800,000 blocks of abstract storage state space to the Arobject File-System at level two, Disk-System can pair Disk-1 and Disk-2 to provide an abstract state space of 100,000 block "reliable storage" that is able to survive a single disk failure. Arobject Disk-System uses transactions Allocate-Blocks and Deallocate-Blocks at level one to produce transactions Allocate/Deallocate-Ordinary-Blocks and Allocate/Deallocate-Reliable-Blocks as primitive steps for level two. Within the state space provided by Disk-System, the File-System at level two can define its own consistency constraints such as dividing the space into system space and a set of user spaces at level three. Within a user's space, one can define his own Arobject such as queues, stacks and records. Users can continue the layering of their own Arobjects.

Despite the conceptual development of Arobject, there is still a number of issues that need to be addressed in both the development and formalization of this new model of computation. Currently, Tokuda, Clark and Locke are developing and implementing their Arobjects and the ArchOS decentralized operating system, and

we are in the process of developing a formal model of the concurrency control and failure recovery rules in the context of distributed abstract data types such as Arobjects.

References

[Allchin 82] Allchin, James E. and Martin S. McKendry.
Object Based Synchronization and Recovery.
Technical Report GIT-ICS-82/15, School of Information and Computer Science, Georgia Institute of Technology, 1982.

[Beeri 83] Beeri, C., P. A. Bernstein, N. Goodman, and M. Y. Lai.
A Concurrency Control Theory for Nested Transactions.
ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, 1983.

[Bernstein 79] Bernstein, Phillip A., David W. Shipman, and Wing S. Wong.
Formal Aspects of Serializability in Database Concurrency Control.
IEEE Transactions on Software Engineering : pages 203 - 216, 1979.

[Bernstein 83] Bernstein, P. A., Goodman, N. and Hadziacos V.
Recovery Algorithms for Database Systems.
Technical Report, Aiken Computation Laboratory, Harvard University, March 1983.

[Eswaran 76] Eswaran, K. P., J. N. Gray, R. A. Lorie and I. L. Traiger.
The Notion of Consistency and Predicate Lock in a Database System.
CACM, 1976.

[Goree 83] Goree, J. A. Jr.
Internal Consistency of A Distributed Transaction System with Orphan Detection.
Master's thesis, Laboratory for Computer Science, M.I.T., 1983.

[Jensen 83] Jensen, E. Douglas.
The Archons Project: An Overview.
Proc. International Symposium on Synchronization, Control, and Communication in Distributed Systems, Academic Press, 1983.

[Jensen 84] Jensen, E. D., Pleszkoch, N.
ArchOS: A Physically Dispersed Operaing System --- An Overview of Its Objectives and Approach.
IEEE Distributed Processing Technical Committee Newsletter, Special Issue on Distributed Operating Systems, June, 1984.

[Korth 83] Korth, H. F.
Locking Primitives in a Database System.
JACM, Vol. 30, No. 1, January, 1983.

[Kung 79] Kung, H. T. and C. H. Papadimitriou.
An Optimal Theory of Concurrency Control for Databases.
In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 116-126. ACM, 1979.

[Lynch 83a] Lynch, N. A.
Concurrency Control for Resilient Nested Transactions.
Proc. 2nd SIGACT-SIGMOD Conf. on Principle of Database Systems, 1983.

[Lynch 83b] Lynch, N. A.
Multi-level Atomicity - A New Correctness Criterion for Database Concurrency Control.
ACM Transaction on Database Systems, Vol. 8, No. 4, December, 1983.

[Molina 83] Garcia-Molina, H.
Using Semantic Knowledge For Transaction Processing In A Distributed Database.
ACM Transaction on Database Systems, Vol 8, No. 2, June, 1983.

[Moss 81] Moss, E.
Nested Transactions: An Approach to Reliable Distributed Computing.
Technical Report, M.I.T. Laboratory for Computer Science TR 260, Apr. 1981.

[Schwarz 82] Schwarz, Peter M. and Alfred Z. Spector.
Synchronizing Shared Abstract Types.
Technical Report CMU-CS-82-128, Department of Computer Science, Carnegie-Mellon University, 1982.

[Sha 83] Sha, Lui, E. Douglas Jensen, Richard F. Rashid, and J. Duane Northcutt.
Distributed Co-operating Processes and Transactions.
Proceedings of ACM SIGCOMM symposium, 1983.

[Silberschatz 80] Silberschatz, A., and Z. Kedem.
Consistency in Hierarchical Database Systems.
JACM 27:1, 1980.

[Tokuda 85] Tokuda, H., Clark, R. K. and Locke, C. D.
Archons Operating System (ArchOS) --- Client Interface, Part II.
Technical Report, Department of Computer Science, Carnegie-Mellon University, 1985.

[Weihl 84] Weihl, W. E.
Specification and Implementation of Atomic Data Types.
PhD thesis, Massachusetts Institute of Technology, 1984.

[Wood 80] Wood, W. G.
Recovery Control of Communication Processes in a Distributed System.
Technical Report, Technical Report 158, University of Newcastle upon Tyne, 1980.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.