FILE COPY

ESD-TR-77-250

ESD ACCESSION LIST

DRI Call No. 87923

Copy No. 1 of 1 cys.

A LAYERED VIRTUAL MEMORY MANAGER



Massachusetts Institute of Technology
Laboratory for Computer Science (formerly Project MAC)
Cambridge, MA 02139

May 1977

Approved for Public Release;
Distribution Unlimited.

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
HANSCOM AIR FORCE BASE, MA 01731

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY
1400 WILSON BOULEVARD
ARLINGTON, VA 22209

ADA046613

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

LEGAL NOTICE

When U.S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

OTHER NOTICES

Do not return this copy. Retain or destroy.

This technical report has been reviewed and is approved for publication.

William R Price

WILLIAM R. PRICE, Captain, USAF
Techniques Engineering Division

Roger R. Schell

ROGER R. SCHELL, Lt Col, USAF
ADP System Security Program Manager

FOR THE COMMANDER

Frank J. Emma

FRANK J. EMMA, Colonel, USAF
Director, Computer Systems Engineering
Deputy for Command & Management Systems

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: ESD-TR-77-250	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: A LAYERED VIRTUAL MEMORY MANAGER	5. TYPE OF REPORT & PERIOD COVERED
6. PERFORMING ORG. REPORT NUMBER: MIT/LCS/TR-177	
7. AUTHOR(s): Andrew Halstead Mason	8. CONTRACT OR GRANT NUMBER(s): F19628-74-C-0193 ARPA Order No. 2641
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Massachusetts Institute of Technology Laboratory for Computer Science (formerly Project MAC) Cambridge, MA 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: A023
11. CONTROLLING OFFICE NAME AND ADDRESS: Deputy for Command and Management Systems Electronic Systems Division Hanscom AFB, MA 01731	12. REPORT DATE: May 1977 13. NUMBER OF PAGES: 133
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for Public Release; Distribution Unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Multics
Virtual Memory Manager
Logical Memory
Physical Memory
Resource Control

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This thesis presents a specification for the Multics virtual memory manager. The virtual memory manager is that part of the operating system which coordinates the usage of physical memory and which manages the bindings between logical memory and physical memory.

The specification is in the form of a model, using the methodologies of type extension and layers of abstraction.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

MIT/LCS/TR-177

A LAYERED VIRTUAL MEMORY MANAGER

ANDREW HALSTEAD MASON

May 1977

This research was supported in part by Honeywell Information Systems Inc., and in part by the United States Air Force Information Systems Technology Applications Office (ISTAO) and the Advanced Research Projects Agency (ARPA) of the Department of Defense of the United States under ARPA Order No. 2641, which was monitored by ISTAO under Contract No. F19628-74-C-0193.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE
(Formerly Project MAC)

CAMBRIDGE MASSACHUSETTS 02139

A LAYERED VIRTUAL MEMORY MANAGER*

by

ANDREW HALSTEAD MASON

ABSTRACT

This thesis presents a specification for the Multics virtual memory manager. The virtual memory manager is that part of the operating system which coordinates the usage of physical memory and which manages the bindings between logical memory and physical memory. In the case of Multics, physical memory is composed of fixed-length blocks called frames and logical memory consists of segments, representing sets of frames.

The original specification is out of date and obsolete because it describes an overly complicated structure and ignores the issue of resource control. The specification described here compatibly updates the functionality of the Multics virtual memory manager, simplifies the requisite structure, and addresses resource control problems.

The specification is in the form of a model, using the methodologies of type extension and layers of abstraction. These methodologies provide the tools to develop a precise model structure, which is capable of handling the intricacies of resource control. The end result is organizational simplicity, certifiability, and comprehensibility.

THESIS SUPERVISOR: David D. Clark

TITLE: Research Associate in the Department of Electrical Engineering and Computer Science

*This report is based upon a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, on May 12, 1977 in partial fulfillment of the requirements for the degrees of Master of Science and Electrical Engineer.

ACKNOWLEDGMENTS

My greatest indebtedness is to Dr. David Clark for the insight and help he gave me throughout the work on this thesis and for the many hours he spent reading my drafts.

Many thanks go to Prof. Jerome Saltzer, for the suggestions and ideas that he provided me.

I want to especially thank Prof. David Redell for starting me on this work and for sharing many discussions with me about the material.

Bernie Greenberg, of Honeywell Information Systems Inc., and Dick Bratt, formerly of Honeywell Information Systems Inc., were very helpful by giving me advice on some of the technical aspects of Multics and my work.

I also want to thank the members of the Computer Systems Research Division of the M.I.T. Laboratory for Computer Science for their support, advice, and comments on my work and my English. In particular, Phil Janson and Doug Hunt were most helpful in discussions on the merits of different aspects of my thesis, and Dave Reed, Bob Mabee, and Andy Huber provided much technical expertise.

This research was supported in part by Honeywell Information Systems Inc., and in part by the United States Air Force Information Systems Technology Applications Office (ISTAO) and the Advanced Research Projects Agency (ARPA) of the Department of Defense of the United States under ARPA Order No. 2641, which was monitored by ISTAO under Contract No. F19628-74-C-0193.

TABLE OF CONTENTS

ABSTRACT 3

ACKNOWLEDGMENTS 4

TABLE OF CONTENTS 5

LIST OF FIGURES 8

Chapter One Introduction 9

1.1 The Problem 11

1.2 Method of Solution 13

1.3 Related Research 15

1.4 Plan of the Thesis 17

Chapter Two The Multics Virtual Memory Manager 19

2.1 The Multics File System in Brief 20

2.2 Some Details on Segmentation and Paging in Multics 25

2.3 Problems with the Current Virtual Memory Manager 36

2.3.1 Page Faults on the FSDCT 37

2.3.2 A Peek at the Quota Problem 38

2.3.3 Conclusion 39

2.4 Summary 41

Chapter Three A Three Layer Virtual Memory Manager 43

3.1 Modularization 44

3.2 Modularizing the Virtual Memory Manager 48

3.3 Ordering the Modules 52

3.4 Objects and Type Managers 56

3.5 Summary 61

Chapter Four The Paging Manager 63

4.1 PAGE CONTAINER Attributes 64

4.2 PAGE CONTAINER Operations 66

4.3 Dependencies in the Paging Manager 68

4.4 Discussion 69

4.5 Extensions to the Paging Manager 72

4.6 Further Thoughts 74

4.7 Summary 75

Chapter Five Resource Control 77

5.1 QUOTA CELLs 79

5.1.1 QUOTA CELL Attributes 79

5.1.2 QUOTA CELL Operations 80

5.1.3 Dependencies in the QUOTA CELL Manager 84

5.2 The PAGEMENT Manager 86

5.2.1 PAGEMENT Attributes 86

5.2.2 PAGEMENT Operations 88

5.2.3 Dependencies in the PAGEMENT Manager 94

5.3 How PAGEMENTs and QUOTA CELLs Fit Together 95

5.4 Resource Control and PAGE CONTAINERs 96

5.5 Summary 99

Chapter Six Segment Support 101

6.1 Active Segments 102

6.1.1 Information in the AST 103

6.1.2 Splitting Up the AST 104

6.1.3 Active Segments and PAGEMENTs 106

6.2 Functions of Segment Support 107

6.3 Summary 110

Chapter Seven Conclusion 111

7.1 Results 112

7.2 Differences between Multics and the Model 114

7.2.1 Visible Functional Differences 114

7.2.3 Invisible Structural Differences 116

7.2.3 Philosophical Differences 118

7.3 Resource Control 119

7.4 Directions for Future Research and Development 122

APPENDIX 125

REFERENCES 129

LIST OF FIGURES

Figure II-1 A Sample Directory Tree 21

Figure II-2 Quota Cells in the Hierarchy 23

Figure II-3 The Hardware Addressing Mechanism 28

Figure II-4 Structure of the Multics Virtual Memory Manager 40

Figure III-1 A Possible Model Structure 53

Figure III-2 Structure of the Model 55

Figure V-1 Moving Quota with Segments 83

Figure VII-1 Final Structure of the Model 113

Chapter One

Introduction

As computer systems find more and more applications, the need grows to guarantee certain properties about them. For example, some worth-while properties concern the security of information stored in a computer and the integrity of the names used to reference the information. Attempts to prove the validity of such properties demonstrate the importance of the operating system, because all of the system software relies on the operating system. Therefore, there is a need to certify an operating system, meaning to guarantee that the operating system matches its specifications. Intuitively, it should be clear that given two systems supposedly having the same functionality, it is easier to certify the one which is simpler. Thus, as a prelude to the certification of a system, the system should be simplified as much as possible. This thesis addresses the question of the simplification of one part of an operating system.

There are few tests or criteria for determining the degree of simplicity of an operating system. About the best test is to assign a competent person to study the code of some subsystem for a few hours. The system is too complex if the person cannot understand how some subsystem works after such study. This test provides a threshold over which a system is too complex, but provides no method for engineering a system below the threshold. What, then, is the nature of complexity? The key to unraveling complexity is structure

[Simon, 1962; Liskov, 1972b]. The better a system is structured, the less complex it is.

1.1 The Problem

The Multics Security Kernel Design Project, of which this thesis is a part, is an effort to redesign the Multics supervisor. The goals of the project are to simplify the system to increase its security and reliability. Some of the work included in the project has been a study of virtual memory mechanisms by Philippe Janson [1976] and a redesign of traffic control by David Reed [1976]. This thesis draws heavily from their work.

The goal of this thesis is a specification of virtual memory management for the Multics system. A specification for a system is a description of its operating characteristics. Although a specification can take many forms, a complete specification dictates the behavior of the system in every situation. A specification is needed for the Multics virtual memory manager because none exists which accurately reflects the current functionality. When Multics was first designed, much thought was given to the specification of virtual memory. One of the hard problems was the design of a subsystem to control and account for the usage of the virtual memory. This subsystem was called resource control. No solution for resource control was found at that time, so its specification was omitted. Later, a resource control mechanism was invented. Since the system was then being implemented, resource control was simply added on to the existing virtual memory manager without updating the specifications. The result was an example of functional entanglement, meaning that the functions of virtual memory management were poorly distributed among the modules of the virtual memory manager. The virtual memory manager became difficult to understand, both because the modules interacted in complex ways, and because

these ways were not reflected by their specifications. This situation has persisted to this day. The problem is that resource control represents a completely different dimension of virtual memory management. It cannot be added in a simple way; to achieve cleanliness and simplicity of structure, it must be incorporated into the design from the start.

In essence, the problem attacked by this thesis has two facets. The first, and more important, is that the specification of the virtual memory manager is incomplete. It does not address the area of resource control. The second is that the implementation does handle resource control, but does so in a confusing manner.

1.2 Method of Solution

The specification proposed here will be presented in terms of the extended type methodology. Some of the related research will be mentioned in section 1.3. As suggested by the introduction, one of the important features of the specification is its structure. It is layered, in the sense used by Dijkstra [1968a], and is composed of type managers. We use this structure for two reasons. First, a layered arrangement of extended type managers is quite precise. This avoids any ambiguity in the specification. Second, a layered structure has important implications for system certification. Rather than forcing a proof for the entire system at one time, each layer can be proved independently. The entire system is proved by a kind of finite induction as follows: Suppose the system is constructed from n layers. The first and lowest layer is a subset of the hardware. Proving the lowest layer forms the basis of the induction. Layer i is proved by assuming the correctness of layer i-1 and then matching the specification of layer i with its implementation. This process is repeated for each layer.

Another implication is that the layers do not have to be proved in any particular order. The proof of layer i does not require the correctness of layer i-1, only the assumption of correctness. Of course, the entire system is not proved until all layers have been proved.

These implications can be utilized because layering requires strong assumptions about the dependencies among the layers. In effect, layer i may directly depend only on layer i-1. It may not use or depend in any way on any higher layer or on any layer lower than i-1. In chapter three, we shall

define dependency precisely and explore some of the implications of the definition.

By using type managers, the specification is a model for the virtual memory manager. Although a specification need only describe the external characteristics of the system, the use of type managers also abstracts the internal implementation. By so doing, comparison of the specification to the implementation is made easier.

1.3 Related Research

The research reported here is based on work in several different areas of Computer Science. These include: modularity and layering, type extension, and program verification, as they apply to operating systems design.

The concept of layers of abstraction originated with Dijkstra and the design of the "THE" system [1968a]. Subsequent systems which expand upon these ideas include the CAL system [Lampson and Sturgis, 1976; Sturgis, 1976] and the Venus system [Liskov, 1972a]. Parnas [1972a; 1972b] has studied general principles of modularity for systems. Recently, he combined his work with the layering approach to describe the design of a family of systems [Parnas, 1976].

Type extension began in the design of languages such as SIMULA [Dahl, Dijkstra, and Hoare, 1972] and ALGOL-68. Liskov also used it extensively in CLU [Liskov et al., 1977]. Janson [1976] extended these ideas to be more flexible in operating systems applications. Type extension was used in systems design for HYDRA [Wulf et al., 1974]. Robinson and others at SRI produced a specification for a layered, object-based system [Robinson et al., 1975]. These are all software efforts; as yet, no one has designed objects into the hardware.

Program verification also started in the field of languages. Naur [1966], Floyd [1967], and Hoare [1969] defined correct operation of a program in terms of assertions about the program and were able to prove assertions about small programs.

Currently, the verification of operating systems is receiving much attention. A methodology supporting proofs of correctness is being developed at SRI [Robinson et al., 1975]. At M.I.T., the Computer Systems Research Division of the Laboratory for Computer Science is in the final stages of a design project to facilitate hand verification of the Multics system by identifying those mechanisms required to guarantee the system's security [Schroeder, 1975]. This project involves the restructuring of the supervisor. As part of the project, Janson [1976] and Hunt [1976] studied alternatives to the virtual memory implementation, and Huber [1976] used separate processes to simplify the structure of the demand paging module.

1.4 Plan of the Thesis

Chapter two presents an overview of the Multics storage system and explores some of its details. Although understanding of this chapter may require careful reading, two rewards can be offered. The first is that several of the problems inherent in the organization of the current system will be immediately illustrated. The second is that a sufficient technical background will be accumulated to understand the motivation for several features of the specification. The Multics virtual memory manager is complex. This is why a specification is needed.

In chapter three, we develop the intellectual underpinnings of the model. We start with several conjectures about how to modularize a given system. These are integrated and applied to the Multics virtual memory manager. Next, the notion of layering is examined and found applicable. The result is a virtual memory manager having three layers. Finally, we give a brief introduction to type managers and the notation (essentially PL/I subroutine calls) to be used in succeeding chapters. The placement of material in chapter three is somewhat anomalous. This material was in fact written after the model was developed, and evolved from some introspection. We decided to place it in chapter three to give insight into the construction of the model before we presented the details.

Chapters four, five, and six develop the model itself, from the bottom upwards. Chapters four and five give precise formulations, in our notation, of the lower two layers of the model. Chapter six discusses the top layer in

general terms. We could not formalize the material in chapter six because of interactions with other parts of the system.

Chapter seven presents a summary of the thesis. General features of the model are recapped. Next, we explore how the model, as presented, differs from the current Multics system. Finally, unsolved problems are discussed, along with suggestions for further research. The Appendix following chapter seven gives a brief summary of the model.

Chapter Two

The Multics Virtual Memory Manager

In this chapter, the Multics virtual memory manager is examined. The chapter is divided into three sections. The first describes the environment and context in which the Multics virtual memory manager exists. The second explains some of the more technical details in the implementation of the Multics virtual memory manager. A reader who is familiar with Multics may skip these sections without loss of continuity. The third section examines some of the problems in the Multics virtual memory manager which will be addressed by succeeding chapters.

2.1 The Multics File System in Brief

Before examining the details of the Multics virtual memory manager, some higher level context is necessary. Multics supports a file system which is organized as a tree hierarchy of directories and segments. A representative sample tree is shown in figure II-1. In the figure, circles represent directories, rectangles stand for segments, and arrows illustrate the hierarchical nature of the tree. The top-most directory is named the ROOT because it is the root node of the directory tree. The directories USERS and LIBRARIES are immediately inferior to the ROOT. By convention, immediate inferiors are called sons and superiors are called parents. Other familial relation names are used to describe other relationships (e.g. brother). The ROOT is the only directory or segment in the hierarchy that does not have a parent. Thus, the parent relation imposes a partial ordering on all elements of the hierarchy, and the ROOT is the supremum of the tree. For more detail, the interested reader may want to examine some of the Multics literature [Organick, 1972; Bensoussan, Clingen, and Daley, 1972].

Directories are simply catalogues. They may contain segments, links, or other directories. Segments hold information, which can typically be programs, data, or text. A link is a named pointer to another element in the file system tree. Thus, directory Jones could contain a link to the segment FORTRAN which is in the directory COMPILERS.

Segments use storage in 1024-word blocks called pages. Associated with each segment is the number of pages that it uses. This is known as the segment's length. Users are charged for the storage used by their segments,

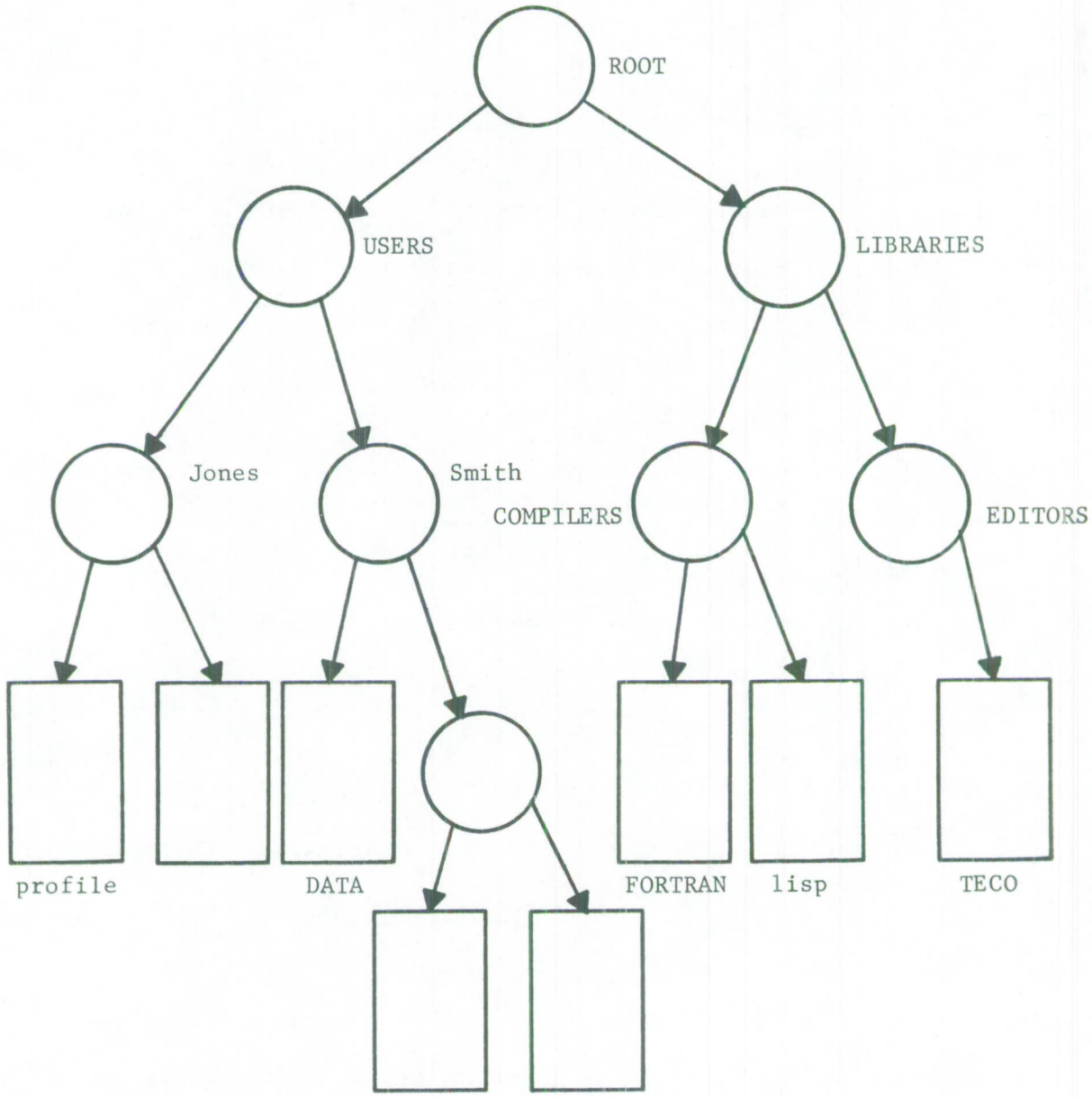


Figure II-1 A Sample Directory Tree

where the charge for a segment is supposed to reflect the amount of secondary memory allocated to the segment over some period of time. However, not all pages of a segment require secondary memory. A very common occurrence is for all 1024 words of a page to hold the value zero. In this case, the page is called a zero page. By not allocating space for zero pages on secondary

memory, the space can be made available for other, non-zero pages. This policy imposes some restrictions on the operating system which will be discussed later. For the purposes of this section, the policy implies that associated with each segment must also be kept the number of its pages which are non-zero, or, equivalently, the number of 1024-word blocks of secondary memory which are actually allocated to the segment. This number is called frames used.

Storage charges are accumulated in a set of designated directories which are called quota directories. Each quota directory holds a quota cell for this purpose. Note that not all directories have quota cells. However, a directory may have a quota cell only if its immediate parent has one (except, of course, the ROOT, which does have a quota cell). Thus, quota directories are also organized into a hierarchy, which is a connected subset of the directory hierarchy. To distinguish between them, the hierarchy of all segments, directories, and links is called the directory hierarchy, and the hierarchy of quota directories is called the quota cell hierarchy.

Within a quota cell is kept the sum of the frames used of all segments charged to the quota cell. Segments in the file system are charged to the most immediate parent directory which has a quota cell. In figure II-2, the segments below directory B are charged to the quota cell in directory B, but the segments below directory C are charged to the quota cell in directory A because C has no quota cell. Note that the quota cell is considered a part of a directory, not inferior to it. Since segments can grow and shrink dynamically, the total number of frames used in the quota cell must be integrated

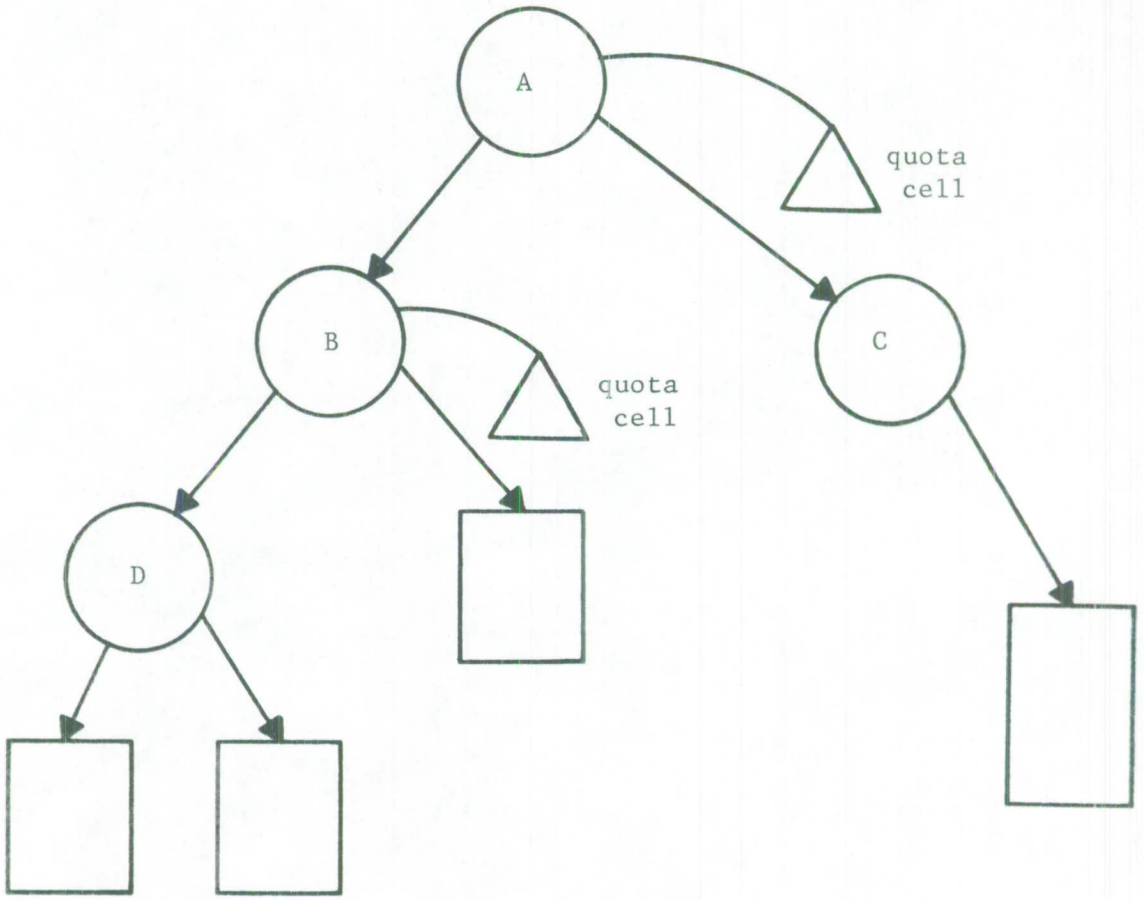


Figure II-2 Quota Cells in the Hierarchy

over the accounting period (typically a month) to calculate the proper amount of storage charged against the quota cell during the period.

To limit the charges for some quota cell, each cell also has a number, called quota, which is the maximum value that frames used may attain. Therefore, whenever a segment is grown, the quota and frames used of the appropriate quota cell must be checked and updated to see if more storage may be allocated for the segment. A quota cell is consistent if the values of frames used and quota are non-negative integers such that $0 \leq$ frames used $\leq$ quota.

The quota cell hierarchy can be modified in two ways: quota cells can be created or deleted and quota can be moved from one quota cell to another.

These operations can be performed by any user having sufficient authority, meaning any user having modify permission to the affected directories. No modification may be performed which would leave any quota cell inconsistent.

2.2 Some Details on Segmentation and Paging in Multics

The basic unit of virtual memory on Multics is the segment. A segment is a variable-length array of words. It has a length, a maximum length, (1) and uses some number of secondary memory frames (see the next paragraph). To reference a word of memory, a two-component address must be specified. The first is the segment number, which uniquely identifies a segment within a process. The second component is the offset. This indicates the proper word within the segment.

To simplify the management of physical memory, segments are broken up into fixed-length pages. At the same time, the physical storage devices (e.g. disk packs) are partitioned into frames, which are the same size as a page. The supervisor moves the pages of the various segments among the frames as required. For reliability reasons, all of the pages of a segment are permanently stored on the same physical device (physical volume). A Volume Table Of Contents (VTOC) is maintained on each physical volume. It contains one entry (VTOCE) for each segment stored on the device. Physical volumes are grouped into logical volumes. A logical volume may contain one or more physical volumes. Since a user may own a disk pack, the logical volume concept provides a way to distinguish among system storage and storage owned by different users.

---

(1) The maximum length of a segment can be changed by calling the supervisor. At all times, the maximum length of a segment must be less than or equal to a system-defined maximum length and greater than or equal to the segment's length.

The operating system remembers the physical location of every page by keeping a page descriptor, or page table word (PTW), for each. The PTW's of the pages of a segment are grouped together, in sequence, to form a page table. The page table, along with other information about a segment, is permanently stored in the segment's VTOCE. In order for the hardware to access a word, the segment containing the word must be active. When a segment is active, its page table, and some of the other information in its VTOCE, is kept in primary memory in a data base called the Active Segment Table (AST).

The concept of the home of a page appears throughout the Multics supervisor. It refers to that secondary memory frame on the physical volume in which the page is permanently stored. When a segment is made inactive (deactivated), all pages of the segment are returned to their homes.

Associated with each process is a special segment called a descriptor segment. This segment contains an array, indexed by segment number, of Segment Descriptor Words (SDW's). A segment is assigned an SDW by the address space manager (see below). If a segment is active and the process has referenced the segment since its activation, its SDW contains the address in primary memory of the segment's page table and the access privileges which that process may exercise on the segment. Such an SDW is said to be connected. If the segment is not active or the process has not referenced it since activation, a flag in the SDW is set. If the flag is set, it means that the SDW must be connected before any reference to the segment may be completed. Note that a segment may have only one page table but an SDW in each of several processes.

The descriptor segment also has a page table and, when the process is running, its address is kept in a special processor register called the Descriptor Base Register (DBR). A process can be executing only if the page table of its descriptor segment is in primary memory (i.e. the descriptor segment must be active). In fact, all descriptor segments are always active.

The hardware addressing mechanism (see figure II-3) works as follows: It is supplied with a two-component address, such as <i,j>. The DBR is used to find the descriptor segment page table. Next, the value of i is divided by the number of SDW's on a page to determine which page of the descriptor segment holds the SDW of the segment. The hardware then reads the SDW to locate the page table of the segment. If the segment is not connected, a flag has been set in the SDW. When the flag is set, a processor exception, called a segment fault, occurs. The fault causes a trap into the supervisor so that the segment can be activated, if necessary, and connected.

When the page table has been found, j is divided by the size of a page in words. The quotient is used to index into the page table to find the address of the proper page. If the page is not in primary memory, a flag in the PTW has been set, which causes a different processor exception, called a page fault. The fault invokes the supervisor to read in the page. Finally, the page is in primary memory and the word is referenced.

For completeness, it should be mentioned that the pages of a descriptor segment need not always reside in primary memory. The page table of a

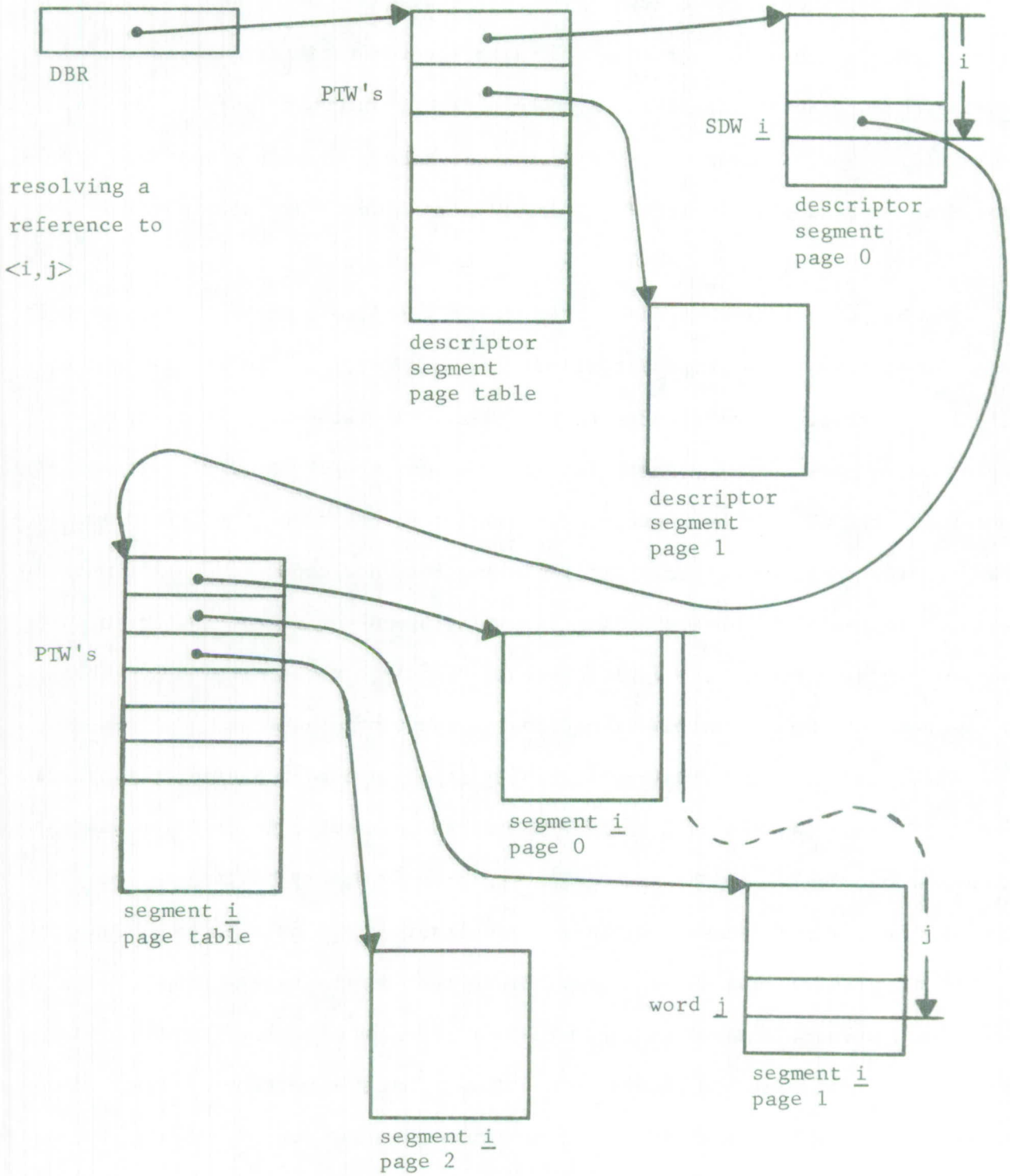


Figure II-3 The Hardware Addressing Mechanism

descriptor segment is like any other page table; if a page of the descriptor segment is not in primary memory, the flag has been set in the PTW and a reference to the page will cause a page fault.

The supervisor module which resolves page faults is page control. In response to a page fault, page control copies pages into primary memory. In addition to primary and secondary memory, page control may use the paging device. The paging device serves as an intermediate holding station for pages. It is typically smaller and faster than secondary memory but larger and slower than primary memory. Primary memory and the paging device have a very limited amount of space for pages. As part of the task of bringing pages into primary memory, page control also moves pages from primary memory to the paging device and from the paging device to secondary memory. The part of page control which performs page removal is called the page removal algorithm. To avoid conflict among different instances of page control, each instance must lock a mutual exclusion lock called the global page table lock.

It should be stressed that a page is a logical unit of 1024 words of information, whereas a frame or home is a contiguous physical unit of storage. At any one time, a page may be stored in several frames or none, if the page is a zero page. Frames and homes may be allocated or freed, modifying the amount of storage which is used by the segment, but not modifying the information content of the segment.

Conceptually, if no value has been written onto a page, the page is defined to contain only zeroes. Therefore, a process may read from a page even if no process has ever written into it. This feature creates a problem

for the management of segments, namely, what should be done with all of those zero pages? If they are physically stored, valuable space in secondary memory may be wasted. If they are not, they will have to be created at the proper time. The supervisor chooses the latter solution. A special null value is placed in the PTW for the page and the page fault flag is set. When in this state, a page is called a null page. If a page fault occurs on a null page, the fault is called a quota page fault, and page control will construct a frame containing zeroes in primary memory, change the segment length if necessary, and modify the PTW to indicate the location of the frame of zeroes. For reliability, page control will also allocate a secondary memory frame from the correct physical volume to be the home for the page. This sequence of operations is called page creation. Note that the creation of a page does not cause the page to spring into existence; it causes the logical page to have a physical representation.

Conversely, before a page is copied back to secondary memory, page control examines it to see if it is a zero page. If so, page control will delete the zero page by freeing the secondary memory frame, changing the segment length if necessary, and marking the PTW as null. Again, page deletion does not terminate the existence of the page; it eliminates the physical representation of the page. Zero and null pages can also occur if some process writes into a page, making the page contain only zeroes. Note that either a read or a write operation may cause the length of a segment to change.

Secondary memory frames are allocated and freed using the data base known as the File System Device Control Table (FSDCT). This data base contains one entry for each frame of each physical volume configured into the system. The

entry consists of one bit, indicating whether the corresponding frame is allocated or freed.

The supervisor module which handles segment faults is segment control. It is responsible for activating and connecting segments on demand and controls the AST data base. Because there is a limited amount of space in the AST for page tables, activating one segment usually requires the deactivation of another. Segment control also handles deactivations. To conserve space in the AST, the page tables of active segments are grouped according to their size. The size of a page table in the AST is determined by finding the smallest power of four that is greater than or equal to the length of the segment. (1) An active segment can grow whenever page control adds zero pages to it. If an active segment grows too much, it needs a larger page table. The size of the page table is stored in the SDW. If an attempt is made to reference a page for which there is no PTW, the hardware will detect this and cause a bound fault. Segment control resolves a bound fault by allocating a larger page table.

Naturally, some segments must remain permanently in primary memory. The most obvious example is the segment which holds page control itself. Since page control directs paging, if it were not in primary memory, it could not be executed, and paging would cease. Therefore, segments such as page control and the AST are implemented from special unpaged segments. An unpaged segment

(1) In a VTOCE, a full-length version of the page table is stored. A shorter page table can be placed in the AST only if the last pages of the segment are null.

has no page table. This fact is indicated by the state of a flag in the SDW, and the processor, upon finding such a segment, knows not to look for a page table. Instead, the address in the SDW is the absolute address, in primary memory, of the first word in the segment.

Similarly, some paged segments, such as segment control, should not be deactivated. If segment control were deactivated, there would be no executable program that could activate segments. Consequently, segment control is an example of an _always active segment_. This is shown by the state of a _hold flag_ in the AST entry (ASTE) for segment control. Segment control knows that if the hold flag is set, the segment must not be deactivated.

The _file system_ is responsible for the permanent storage of segments. At this level, segments are the logical nodes of a directory tree. One of the purposes of the file system is to provide a convenient, user-oriented, global name space for segments. In addition, the file system maintains segment attributes. Some example attributes are the segment's access control list (ACL), its maximum length, the date and time it was last modified, the identifier of the physical volume on which it is stored, and the address of its VTOCE. The file system provides the ability to create or delete segments and list or modify their attributes. Directories are special extended-type objects, implemented by segments, which may be examined or modified only through calls to the supervisor.

The file system must also maintain the quota cell hierarchy. The function of the quota mechanism includes the ability to move quota between a quota cell and one of its immediate inferiors and the ability to create and delete

quota cells. When these operations occur, the file system must make the appropriate modifications to the quota cell tree.

A segment, identified by its file system name, can be assigned a segment number through a call to the address space manager. When a segment is associated with a segment number, it is said to be known to the process. The address space manager also provides facilities to terminate or revoke the segment number-file system name binding.

The quota mechanism regulates the amount of storage allocated to the subtree under a given directory. One unit of quota corresponds to the ability to use one secondary memory frame. Every time a zero page is created from a null page, page control must check the appropriate quota cell, which is part of a parent directory's VTOCE and ASTE. The checking is expedited by requiring all parent directories of an active segment to be active. In that way, the quota cell is guaranteed to be available in primary memory if needed. Unfortunately, this also ties up primary memory space for quota cells which are rarely needed. Each ASTE contains a pointer to the ASTE of the segment's parent directory. Page control finds the proper quota cell by stepping through the chain of parent directories until a directory is found which contains a quota cell. By definition, this is the quota cell for which page control is searching. Intermediate directories, those between a segment and the directory containing the quota cell against which the segment is charged, do not have quota cells. A quota cell contains the amount of quota that may be used by all segments which are charged against it and the amount that is actually

being used (frames used). At no time can the value of frames used exceed the value of quota. For accounting purposes, a quota cell also contains an estimate of the time-frames used integral charged to that quota cell since the start of the accounting period. If an attempt is made to use more quota than is available from a quota cell, a record quota overflow condition is signaled to the user by the supervisor.

Page control relies on segment control to maintain the correct value of the parent ASTE pointer. Since segment control also depends on page control to implement the demand paging algorithm, there is a dependency loop in the virtual memory manager.

From time to time, it is necessary to revoke the access privileges that a process may exercise on an active segment. This is done by setting the segment fault flag in the SDW. Segment control can differentiate between this kind of segment fault and a normal segment fault because the value stored into the SDW is different in each case. Segment control then reflects an access revocation fault to the proper fault handler.

The concepts of zero page and null page have been discussed earlier in this section. Before going to the next section, one refinement of a similar nature needs to be presented. A semi-null page belongs to an intermediate class of pages which has been introduced in an effort to improve performance. They are relevant to this thesis because they exist now on Multics, and will appear in a slightly modified form in chapter four. A semi-null page represents a page of zeroes. It is not physically stored in secondary memory, but

is associated with a secondary memory home. The system designers feel that a zero page can be created faster than a zero page can be moved from secondary memory. Before removing a page from primary memory, page control checks whether the page is a zero page. If so, contrary to the statement made earlier in this section, the page is transformed into the semi-null state and the home associated with it is not freed. Frames used, however, is decremented. If the page is brought back into primary memory before the segment is deactivated, a zero page can be manufactured without also having to allocate a home on secondary memory. A semi-null page is finally changed into a null page when the segment is deactivated. The advantages of semi-null pages are that they can reduce the number of I/O operations to secondary memory and they can reduce the frequency of allocations from secondary memory. One interesting aspect of semi-null pages is that, by the current definition, they do not use quota. This produces the slightly anomalous situation that a home can be associated with a page and yet not be charged against any quota cell.

2.3 Some Problems with the Current Virtual Memory Manager

Although the description of the Multics virtual memory manager given in section 2.2 is not exhaustive, enough background has been presented to discuss some of the weaknesses in the implementation. In this section, two specific problems with the virtual memory manager will be examined. These are not the only problems, but have been chosen to illustrate the poor modularization of the virtual memory manager. The nature of the problems can be characterized as functional entanglement, meaning that the functions to be performed are poorly distributed among the modules. The modules interact badly, producing complexity and making the system difficult to understand. These problems are typical of the confusion in the implementation of the virtual memory manager. They are examples of the second facet of the overall problem discussed in section 1.1.

In the first example, an artificial recursion is used to handle paging under special circumstances. Besides the fact that the recursion is difficult to understand, the existence of the recursion masks a much simpler solution, which will be presented in chapters four and five. In the second example, the current modularization is shown to be defective because it does not adequately reflect the needs of virtual memory management. An important factor, resource control, is underemphasized.

2.3.1 Page Faults on the FSDCT

A process takes a page fault either to copy a page into primary memory or to create a page of some segment. The actions required are different for each case, but the same module, page control, handles both. If a page is to be created, page control must find a free home for it on the proper physical volume. To do this, page control uses the FSDCT data base. The FSDCT can be quite large, so it is kept in a paged segment. Therefore, page control, the module which handles page faults, must be able to take a page fault on the FSDCT.

Not surprisingly, this is done with a special case mechanism. Page control first checks to see if the needed page of the FSDCT is already in primary memory. If not, page control carefully stores the data about the page fault being processed and calls itself recursively to copy the FSDCT page. The size of the FSDCT cannot be changed by page control, so page control need never try to create a page for the FSDCT (another special mechanism is used). This guarantees that there are no potentially infinite sequences of page faults on the FSDCT. However, page control must be careful not to destroy any data relating to the original page fault.

There is a certain elegance to the idea of using a recursive mechanism to reference the FSDCT. Taken as a whole, however, the mechanism reeks of poor design. Rather than performing a real recursion and faulting on the FSDCT, page control modifies its environment to look as though a fault had been taken. After copying the page of the FSDCT, an artificial return again modifies the environment so that the original page fault can be processed. This

very artificial recursion is extremely hard to decipher. Instead of achieving any economy of mechanism, the recursive use of page control makes the understanding of the virtual memory manager more difficult.

2.3.2 A Peek at the Quota Problem

The term quota problem is used loosely to refer to a large set of complexities in the supervisor. It is an example of functional entanglement on a large scale. Rather than attempt to discuss the entire problem, this section will present one aspect of the quota problem.

The hierarchy of quota cells is dynamic, meaning that it can undergo frequent modifications. Such modifications are the result of requests to the supervisor from users having sufficient authority. Since quota cells are modified by page control when creating or deleting a page, modifications to the quota cell hierarchy must be coordinated with page control.

When a user wishes to change the quota cell hierarchy in some way, the user issues a request to that effect to the supervisor. The supervisor first validates that the user has sufficient authority to request the modifications and then calls the program which modifies the quota cell hierarchy, quotaw, to perform the operations. To avoid any conflict with page control, quotaw first locks the global page table lock. Since both quotaw and page control use quota cells as data bases, locking the lock guarantees that only quotaw can modify the contents of any quota cell. Unfortunately, locking the lock also stops all paging activity in the system. Next, quotaw checks the request to ensure that the quota cells affected will remain consistent after the modifi-

cation. Finally, if the check succeeds, the request is performed and the lock unlocked.

This description seems simple enough, but to what module does quotaw belong? Since quotaw locks the global page table lock, a fair assumption might be that it belongs in the page control module. This would mean that page control is responsible for copying pages, creating pages, and modifying the quota cell hierarchy. That is a very large task to be performed by one module. In addition, the quota cells of active directories are kept in the ASTE's of the directories, which are supposed to be part of a segment control data base. If quotaw is part of page control, it should not manipulate the data bases belonging to other modules. Suppose, on the other hand, that quotaw is not part of page control. Then it violates any semblance of modularity by locking the global page table lock. No matter how segment control and page control are chosen, the existence of quotaw ruins the division between them and makes each dependent on the other.

2.3.3 Conclusion

The Multics virtual memory manager is loosely organized into two modules, segment control and page control. The two modules are pictured in figure II-4. The fact that each module depends on the other is symptomatic of poor modularization. However, it should be noted that the structure does conform to the original specification of the virtual memory manager. When the specification was developed, mutual dependencies, such as those displayed by page control and segment control, were not considered unacceptable. Later advances

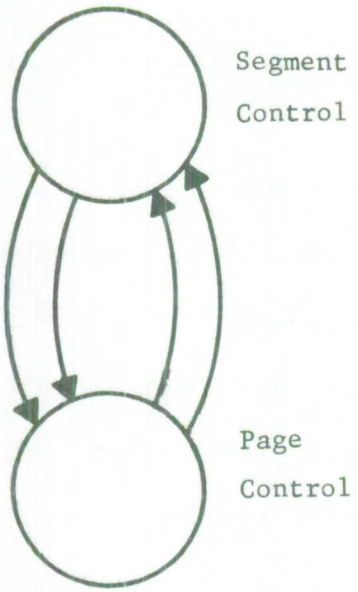


Figure II-4 Structure of the Multics Virtual Memory Manager

in modularization revealed that mutual dependencies led to difficulties in understanding and implementation (see section 1.1).

Examination of the internal operation of each module reveals a clue for how to remedy the problem: Page control is responsible for two separate functions, paging and the control of page resources. Segment control is also responsible for two functions, control of page resources and segmentation. It should therefore not be surprising that the interface between them shows so many interconnections and that many of the interconnections concern resource control. The interface is exactly what could be expected if someone arbitrarily divided a resource control module into two parts and incorporated one part into segment control and the other part into page control.

2.4 Summary

The Multics virtual memory manager and the Multics addressing mechanism have been presented in some detail. Using this foundation, two of the problems found in the current implementation were discussed. The fundamental weakness is the absence of a simple, complete, consistent specification of what the virtual memory manager should implement. The existing specification is not complete in that it does not cover resource control. In the implementation, the existing specification is followed as closely as possible, but resource control cannot be simply added to the virtual memory manager without ruining the modularization. In succeeding chapters, the issue of how to devise a better modularization will be addressed.

Chapter Three

A Three Layer Virtual Memory Manager

In chapter two, discussion centered on how the Multics virtual memory manager is structured and what is wrong with it. Now it is time to address the question of what to do about it. The remainder of this thesis will develop a model of the Multics virtual memory manager. There are two reasons for doing this: one specific and one general. By proposing a model and comparing it to the real system, we can attempt to rectify the drawbacks already outlined. If valid, the model will embody the fundamentals of the virtual memory manager and can serve as a guide to future development and maintenance on Multics. In a larger sense, a model can separate the important issues from the unimportant. In this way, we can learn which considerations should be explored when virtual memory is encountered in a different context.

Section one of this chapter discusses modularization issues of how the model should be constructed. A method will be presented for modularizing a system. In section two, the method will be applied to the virtual memory manager to arrive at a a particular set of modules. Section three discusses why and how the set of modules should be ordered into three layers. Finally, we discuss a particular technique, type extension, for imposing our structure on the current system. This technique will be used in chapters four and five. Section four may be skipped if the reader is familiar with type extension in the context of operating systems.

3.1 Modularization

The concept of modularity has been important in programing for a long time. It grew out of the need to be able to develop and maintain large, complex systems. By breaking large programs into smaller, simpler ones, the system could be written, compiled, tested, and debugged in parallel, thus increasing the productivity of a programing team. Of course, a program cannot be divided arbitrarily, because there is no reason to believe that an arbitrary division would allow the parts to be developed separately. When a system is modularized, the modules have connections to each other in various ways. By connections, we mean the assumptions which the modules make about one another [Parnas, 1971]. If we are not careful, increasing the number of modules may cause the number of connections to grow in a combinatorial explosion. Therefore, while breaking the system into small, simple modules, we must also try to keep the number of intermodule connections to a minimum. The best way to do this is to divide the system along functional boundaries because, intuitively, those boundaries define a partition of the system having a relatively small number of connections.

So far, the terms module and function have been used loosely. This is because they are relative. To an operating system, one function might be virtual memory; but inside of the virtual memory manager, many subfunctions can be seen. Therefore, some techniques are needed which can help differentiate functions, and thus module boundaries, in a given context.

The first technique deals with data base references. If two parts of a system reference mutually exclusive external data, they should belong to different modules. The term external here means data other than arguments. Clearly, both a calling program and its subroutine will reference the arguments passed between them. This technique makes sense because if two parts of the system can be placed in different modules without increasing the connectivity of the system, they should be. The converse is also useful, i.e. if two parts of the system reference the same data bases, they are likely to be parts of the same module. This technique is the strongest because functions are frequently described and thought of in terms of their effects on data.

The second technique is that if one part of the system must depend on a second, but the second does not need the first, then the two parts should be implemented in different modules. Implementing or understanding a single module containing both parts is more difficult than implementing or understanding two separate modules. There are two motivations for this technique. One is that we wish to explicitly recognize dependencies in the system, both for informal certification and to increase our understanding. The other relates to the principle of least privilege [Saltzer, 1974]. If protection barriers are available within the system, they can be used in this situation to ensure that damage to the first part of the system does not easily spread to the second. This implies, for example, that the trigonometric functions should be separated from the floating point package, because floating point operations are needed to calculate sines and cosines but trigonometry is not needed for multiplication.

Third, if there is a function or service common to two or more parts of the system, the common part should be modularized separately. This derives from the idea that there is no need to reinvent the wheel. Rather than force each user to implement his own file system, one is provided for all by the operating system. This has also been called the principle of greatest common mechanism [Hunt, 1976].

The fourth technique derives from the principle of least common mechanism [Popek, 1974; Schroeder, 1975] and, in some respects, is the converse of the second. It says that if one function is common to many users and another is common to only a few, the two functions should be separated. The idea is that the amount of the system on which a module depends should be minimized by placing unneeded functions in a separate module. This technique is not the inverse of the third technique and is, in fact, quite compatible.

The last technique involves the frequency of use. If two pieces of the system operate at different rates, they are likely to be parts of different modules. Consider a system having one user process and one server process. The user process requests two kinds of services, A and B, from the server. Service A is requested once a second. Service B is requested once a minute. Because of the disparity in the rates of the requests, the server process could be divided into two modules. One motivation for dividing the server is to guarantee that service for requests of type A is not impaired by interference with service for requests of type B.

These five techniques are not meant to be exhaustive or definitive. They have been phrased in terms such as _should_ and _likely_ because they are indicators; they can only give clues as to where module boundaries could be placed.

Certainly, situations exist which would yield conflicting or misleading clues. In the process example above, suppose that both services required the same data base. Then the first technique would suggest that one module is appropriate, but the fifth would indicate two. These techniques are advanced to provide something, besides personal bias, as a basis for modularization.

Given a system comprised of only one module, the techniques can be used, iteratively, to approximate the optimum modularity. However, we will not attempt to prove that they can be applied deterministically, or that they are guaranteed to converge to the optimum point. The next step is to return to the Multics virtual memory manager and identify, using these techniques, what parts should be in separate modules.

3.2 Modularizing the Virtual Memory Manager

The Multics virtual memory manager uses four major data bases: the AST, quota cells, page tables, and the File System Device Control Table (FSDCT). (1) The FSDCT contains a list of every secondary memory frame available to the system, along with an indication of whether the frame is allocated for any page. It is used during page creation and deletion. The other three data bases should be familiar from chapter two. Naively, we might think, using technique one, that there should be four modules. However, some of these data bases are used together. Rather than examine the uses of each data base, we shall consider them as one pool of data. This is done for two reasons. First, we wish to start by assuming the virtual memory manager as one huge module. In this way, we can apply the techniques and, hopefully, arrive at a new modularization. Second, we want to allow for the possibility that the current data bases are not divided along functional lines. By looking at all of the data together, we can ignore the effects of the current modularization.

Examination of the virtual memory manager reveals three loci of reference. The first involves only PTW's, which are the elements of page tables, and a few fields in the AST. These data are used when a page is moved from secondary memory to primary memory or back again. We shall call this locus demand paging. The second locus is defined by the data needed for page creation and deletion. It includes the FSDCT, quota cells, page tables, and some fields of the AST. This locus will be called resource control. The data within the AST used for demand paging and resource control is disjoint. The

(1) In the current system, quota cells and page tables are part of the AST.

only overlap occurs on page tables. Careful study of the overlap shows that resource control references page tables to create or delete pages. A page being created or deleted cannot be moved. Thus, although some physical overlap exists, temporal factors ensure that demand paging and resource control never try to use the same data at the same time. The overlap can be conceptually eliminated by having resource control send requests to the demand paging locus to create or delete specific pages.

The remainder of the data represents the bulk of the AST. It is composed of many fields and, correspondingly, has many uses. The principal uses are for the activation and deactivation of segments, for bound faults, and to service external requests originating outside of the virtual memory manager (e.g. moving quota from one directory to another). This locus is called segment support. Remarkably, this locus does not overlap greatly with either of the other two. Some overlap does exist, but that is mostly a consequence of having segment support service requests. For example, a request to deactivate a segment will, of necessity, move some pages onto secondary memory and free a page table. However, only in a very few cases are data belonging to another locus used as decision variables for segment support. This strongly suggests the existence of a natural modularization for the virtual memory manager. Namely, one module for each of the three loci of reference.

The programs which are contained in segment support are paged. This means that they may move freely between primary and secondary memory. This also means that they must depend on that part of the virtual memory manager which handles paging. By technique two, this is confirming evidence that demand paging should be in a separate module.

The FSDCT can be a very large data base. It is so large that is cannot always fit into primary memory. To operate efficiently, the virtual memory manager uses a highly modified form of paging to move needed pages of the FSDCT (see section 2.3.1). With some programing effort, page creation and deletion could use the existing demand paging facility when referencing the FSDCT. This would eliminate a special mechanism and simplify the virtual memory manager. Thus, by the third technique, there is further evidence that demand paging should be separate.

The fourth technique also applies to page movement. In the previous paragraphs, we developed that the demand paging function can be common to both segment support and resource control. This technique also suggests that the management of paging belongs in a separate module.

The last technique provides supporting evidence of three separate modules. Segments are activated at a frequency of about 1.5 times per second, about 3 pages are created each second, and over 100 pages are moved from secondary memory to primary memory every second. (1) Here, again, is strong evidence that demand paging should be separated. Although the frequencies of segment activations and page creations do not differ greatly, we feel that the factor of two difference does suggest the possibility of a module boundary.

In essence, the virtual memory manager performs three identifiable functions for users. First, it is assuming the entire responsibility for demand

---

(1) For our purposes, segments are deactivated at the same rate as they are activated. They are deactivated to make room for a newly activated segment. Similarly, pages are removed from primary memory to make room for other pages, and thus are removed at the same rate as pages are brought into primary memory. However, page creation and deletion are quite distinct. Figures could not be obtained on the rate of page deletion, but it is reasonable to assume that the frequency of page creation or deletion is about 4 per second.

paging. By this, we mean the management of physical volume frames (homes) and extant pages so that the user is freed from worries about the physical location of pages. Second, the virtual memory manager allows the creation and deletion of pages. This second function includes the actual create and delete operations, the mechanisms to automatically invoke creation or deletion when appropriate, and the facilities to control their invocation according to policies specified by higher layers. Third, the virtual memory manager groups pages together to help implement the information containers called active segments, and provides many utility functions for external use.

Put another way, the first function physically manages the set of existing pages. The second function controls how and when the set of existing pages can change. The third function constructs segments out of pages to facilitate their implementation. The evidence dictates that the three functions should be placed in separate modules.

3.3 Ordering the Modules

As seen in chapter two, the Multics virtual memory manager is poorly modularized. The key, then, to curing the observed functional entanglement is to find a better choice of modules which can perform the same task. A module is a responsibility assignment [Parnas, 1972b]. In other words, a module consists of the collection of programs and data bases needed to perform some task.

As was pointed out in section 3.2, the virtual memory manager performs three tasks or functions. As a first step, the virtual memory manager should be partitioned into three modules, not the existing two. Each module should perform exactly one of the virtual memory manager functions. This step is easy to understand, but how should the modules be structured?

From the considerations discussed in section 3.2, a preliminary structure for the virtual memory manager can be constructed. It is shown in figure III-1. The circles represent modules, and the arrows represent the module dependencies. The presence of the arrow labeled A points out an important consideration: Should the resource control module control the interactions between demand paging and segment support? This question will be answered later in this section, after some background is presented.

The technique of layers of abstraction was first introduced by Dijkstra [1968a]. It involves separation of a system into a series of linearly ordered layers, where each layer, consisting of a set of modules, performs a set of related functions. Higher layers may use lower layers, but lower layers may neither use not depend on higher layers in any way. In terms of the connec-

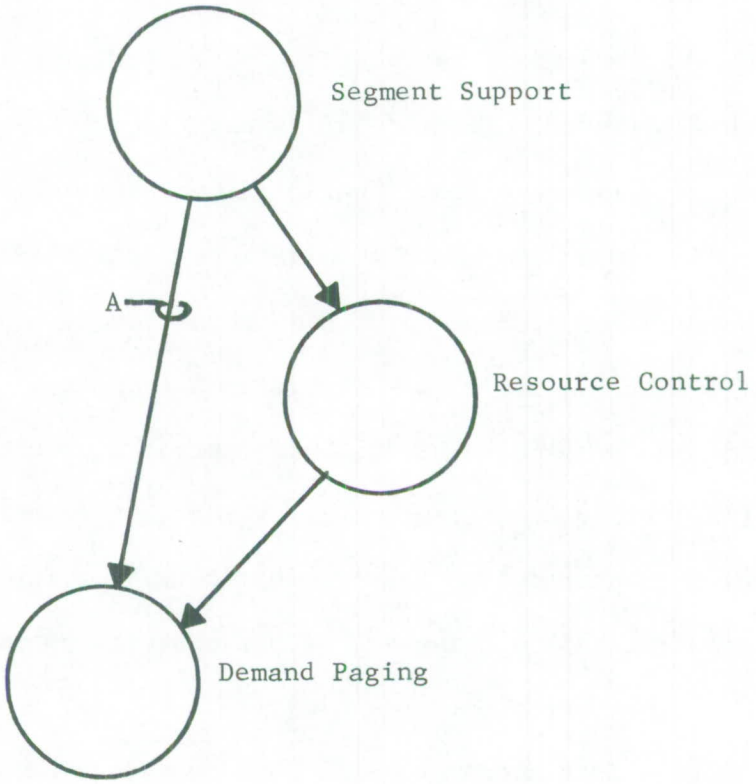


Figure III-1 A Possible Model Structure

tions among modules, the layering technique means that no connection between two modules may pass completely through a layer. If, for example, resource control is supposed to be a complete layer, the arrow labeled A should not be allowed because it by-passes resource control. By rigid adherence to a five--layer structure, Dijkstra was able to design, implement, and debug a medium--scale system in a short time and with very few people. This technique seems to have some attractive properties, but is it applicable here?

The meaning of the term module changes with context. Viewing the operating system as a whole, the virtual memory manager is one module. However, within the virtual memory manager, there are three modules. This suggests

that the system can be thought of as constituting a hierarchy of modules. Each module can be successively divided into smaller modules, until we have only individual machine instructions. In fact, instructions can also be subdivided until the boundaries of particle physics are reached. Simon analyzed systems taken from many disciplines and found the hierarchy concept almost universal [Simon, 1962]. In his view, organizing a complex system as a hierarchy is critical to understand, describe, and control the system.

This kind of hierarchy orders increasingly fine partitions of the system. There is another important module hierarchy, which is the hierarchy of module dependencies. It is related to the graph of the connections among the modules, given a particular partition.

Given that Multics is a hierarchy of module dependencies, Dijkstra [1968b] states the fundamental reason why the hierarchy should be viewed as a series of layers. The reason is that an important function of an operating system is to provide resource allocation. The modules should be ordered into layers to hide the fact that the modules themselves use some of the resources provided by other modules lower in the hierarchy. Otherwise, confusion reigns.

This is directly applicable to the virtual memory manager because, as has been stated, one of its functions is resource control. Thus, the virtual memory manager should be structured into three layers, as in figure III-2. The ordering constraints on the modules, which were developed in section 3.2, still apply, so demand paging should be the bottom layer, resource control should be the middle layer, and segment support should be the top layer. Note, in particular, the absence of the arrow labeled A from figure III-1.

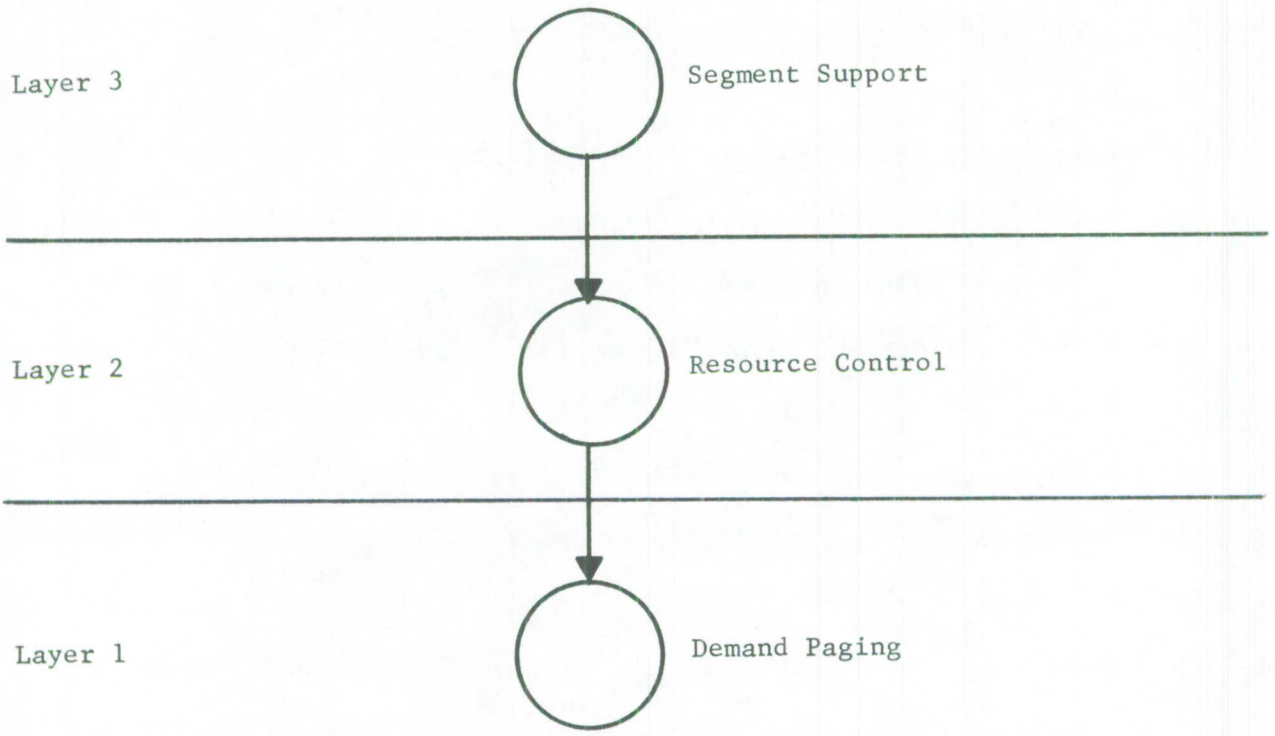


Figure III-2 Structure of the Model

Because the system is layered, all dependence of segment support on demand paging must first be routed through resource control.

A general discussion of modularizing the virtual memory manager is all very well, but specifics are needed. Are there any methods of module description which can explicitly recognize connections? The answer is yes. Type extension is such a description method. The next section will briefly introduce type extension and show why it is useful.

3.4 Objects and Type Managers

Type extension is being used quite extensively in the design of structured programing languages such as CLU [Liskov et al., 1977] and SIMULA [Dahl, Dijkstra, and Hoare, 1972]. However, use of this modeling technique for operating systems is still new. Philippe Janson [1976] has applied type extension to virtual memory mechanisms and David Reed [1976] has used the technique in studying processor scheduling and traffic control.

The type manager construct has several desirable properties that make it attractive for modeling purposes. The objects handled by type managers are completely defined, so there can be no question about the purpose, function, or usage of objects. Second, the interfaces among type managers are well--defined. All communication among type managers must occur openly across the interfaces. Third, the internal representations of objects are completely hidden from other type managers which use them. This ensures that no unforeseen side-effects can occur. Fourth, the dependencies among type managers can be generated in a straightforward manner. Finally, if the objects and their attributes are chosen carefully, their usage will be natural and intuitive. This is important, in operating systems design, to prevent the spread of complexity. In other words, type extension provides a natural way to modularize a system along functional boundaries. This is exactly what is required by section 3.1. These properties are by no means exclusive to type extension, but since type extension has them, it is to our advantage to use type extension in this context.

An object is defined by the set of operations which may be applied to it. It has a set of attributes, which correspond to the properties of the object. One of the attributes of every object is its name, whose value must be unique over the relevant universe of discourse. A type is a set of objects which all have the same set of attributes. All objects of a particular type are managed by one type manager. For example, the objects of type REAL NUMBER would be managed by the REAL NUMBER type manager. They might have attribute sets consisting of the attributes name and value. The name attribute might have values such as x, y, or z, and the value attribute might be -1, 5, or 3.14159.... Hereafter, type names will be given in capital letters to distinguish formal types from the concepts that they are attempting to represent.

Some operations on REAL NUMBERs might be: creation of a REAL NUMBER having the name A and value 3, deletion of A, and addition of the values of X and Y and storing the result in the value of Z. Further operations could be defined so that the type REAL NUMBER corresponds to the mathematical notion having the same name.

More complex types, called extended types, can be defined in terms of already existing types. The representation of an object is the set of objects used by the type manager to implement the object. The map of a type is a data base, internal to the type manager, which indicates the set of component objects which make up the representation of every object of the type. Naturally, any operation defined on objects of some extended type must be expressible as operations on the component objects.

One of the strengths of the type extension modeling technique is the independence of an object from its representation. Users of a type need have

no knowledge whatsoever of how the type is implemented or internally represented. Consider, for example, the extended type VECTOR. Using the type REAL NUMBER, the type VECTOR can be defined to correspond to the mathematical concept of two-dimensional vector. What about the representation of VECTORs? The representation is completely up to the implementer of the VECTOR type manager. VECTORs may be internally represented in either Cartesian coordinates or polar coordinates, using two REAL NUMBERs. The choice will probably depend on the intended or anticipated use of VECTORs, or which representation is more appropriate for the set of operations provided. Conceivably, the manager might use both representations or switch between them as convenient. The manager might even represent them in elliptical coordinates. The point is that the internal representation of a VECTOR is completely irrelevant to a VECTOR user. As long as the user and the type manager agree on a way of communicating about VECTORs, the user does not need to know anything about their representation.

Janson [1976] identified two fundamentally different kinds of types: create/delete (C/D) types and allocate/free (A/F) types. There is essentially an infinite supply of C/D type objects. They are created as needed, used, and then discarded. Most work involving types has concentrated on C/D objects. A/F objects can be neither created nor destroyed. They exist in limited numbers. They are allocated as needed and as available, used, and then freed for subsequent use. The manufacture of A/F objects falls under the category of reconfiguration, which is thoroughly treated by Schell [1971]. In computer system, A/F objects generally have hardware representations and represent some reusable resource (e.g. secondary memory frames).

When using objects for operating system design, five dependencies among type managers can be identified:

(1) Component -- the type manager for type A is dependent upon the type manager(s) which provides the memory space in which objects of type A are stored.

(2) Program -- type A type manager depends on the type manager(s) which provides the memory space in which the programs implementing the type A type manager are stored.

(3) Map -- type A type manager depends upon the type manager(s) which provides the memory space in which the map and other data bases are stored for the manager of type A.

(4) Environment -- type A type manager depends on the type manager(s) which structures the address space or naming environment of programs that implement type A.

(5) Interpreter -- type A type manager depends on the type manager(s) which controls the allocation of processor resources which are used to implement type A.

Put another way, type manager A depends on type manager B if the incorrect functioning of B can cause the incorrect functioning of A. This definition of dependence is intentionally precise and is narrower than the notion of connection. With this definition, we can capture the essence of which types need which other types and discard other kinds of interactions. For example, suppose type manager B performs a service for type manager A. A depends on B, by

this definition. By changing inputs, clearly A can affect the operation of B. However, since a malfunction in A cannot cause a malfunction in B, B does not depend on A. B is connected to A because B certainly assumes that A wants the service performed, and both type managers must reference any arguments passes between them.

Type manager dependencies are transitive in that if type A depends on type B and type B depends on type C then type A depends on both types B and C. In operating systems design, it is important to recognize dependencies to ensure that no two type managers are symmetrically dependent (i.e., depend on each other). Clearly, if two type managers depend on each other, the security, reliability, and understandability of the system is very much in doubt. Therefore, the dependence relation should also be asymmetric and not reflexive. In other words, there should be a partial ordering (i.e. a hierarchy) among type managers which guarantees that no type manager depends upon itself. By examining the maps of the type managers, the complete dependency graph can be generated. Simple inspection will reveal whether the graph does or does not represent a partial ordering. Note that the dependency graph is similar to Parnas's hierarchy of uses [1976].

Note that in section 3.3, discussion centered on a layered structure, whereas in this section, a hierarchy is considered. The use of type managers does not automatically lead to a layered structure. Rather, the use of type managers helps to identify the dependencies within the system. To create a layered system, the designer must still examine the dependencies to check that no mutual dependencies exist and that no dependency by-passes a layer.

3.5 Summary

In this chapter, we developed a method to modularize a system. While the method was quite useful, no statement can yet be made about the optimality or applicability of the method in general. It is at least better than arbitrary choice or personal inclination. The method was applied to the Multics virtual memory manager. The resulting set of three modules needed to be ordered, according to their respective dependencies. Finally, a technique of formally describing a system was introduced, which explicitly recognizes module dependencies. In chapters four and five, this technique will be used to model the bottom two layers of our proposed structure.

Chapter Four

The Paging Manager

In chapter three, the broad outlines of demand paging were defined and we showed that demand paging belongs at the bottom of a dependency graph in the virtual memory manager. This chapter examines in detail the demand paging function. The function will be modeled by a type manager, called the paging manager. The model will then be related to actual system operations, as described in chapter two.

The essential function of the paging manager is to provide PAGE CONTAINER objects to higher layers of the operating system and, ultimately, to the user. PAGE CONTAINERs are designed to store logical pages so that they may be referenced quickly. The mechanics of physical management of PAGE CONTAINERS is completely hidden from users of the paging manager. The number of PAGE CONTAINERs which may be in use at any one time is limited by the size of primary memory: The status and physical location of every PAGE CONTAINER must be maintained in primary memory, and there must be enough room to hold the words of at least two PAGE CONTAINERs. PAGE CONTAINERs are A/F objects, meaning that they always exist, even when not in use.

4.1 PAGE CONTAINER Attributes

The attribute set of a PAGE CONTAINER consists of a name, a data array, (1) a home, a used flag, a modified flag, a zero flag, and a core flag. The name of a PAGE CONTAINER uniquely identifies that PAGE CONTAINER from all other pages. (2) When a PAGE CONTAINER is in the free state, only the name has any meaning; the other attributes may not be referenced. The data array attribute holds the values of the words in the PAGE CONTAINER. The data array is also called the contents of the PAGE CONTAINER. The home attribute refers to the permanent secondary storage location for the logical data contained in the PAGE CONTAINER that the paging manager may use to store the contents of an allocated PAGE CONTAINER. (The use of this attribute will become clearer when PAGE CONTAINER operations are explained.)

The four flag attributes provide auxiliary information about the page held in a PAGE CONTAINER. The used and modified flags tell whether the page has been used or modified since it was allocated. The zero flag indicates whether the data array contains all zeroes. The core flag is used by the seg-

---

(1) In Multics, the number of words contained in a page is 1024. The particular number is not relevant to this discussion, but all PAGE CONTAINERs must contain the same number of words. Those familiar with the history of Multics will recall that the original design called for pages of two sizes, 1024 and 64. In such a design, either another attribute, page size, must be provided, or two paging managers must be used.

(2) There is currently a small controversy over the proper scope of object names. Purists insist that a name must be completely unique over the entire set of objects supported by the system. Given the ability to share information among computers, one can speculate whether the name must then be unique over the objects available to some set of computer systems (this is very much a research topic). On the other hand, more practical designers contend that a name need be unique only over the most relevant domain, e.g., the set of PAGE CONTAINER objects available to a particular computer.

ment deactivation algorithm and tells whether the PAGE CONTAINER is in primary memory.

4.2 PAGE CONTAINER Operations

In the following discussion, operations will be designated like PL/1 call statements. Output arguments will be underlined.

The most fundamental operations are allocate, free, read, and write. The allocate operation can be represented as allocate (home, zero_flag, name). Its function is to select a PAGE CONTAINER from the pool of unused PAGE CONTAINERs and allow it to be used. The value of the home and zero_flag arguments are assigned to the home and zero flag attributes of the PAGE CONTAINER being allocated. If the zero_flag argument is set, then the data array attribute of the PAGE CONTAINER is defined to contain all zeroes, regardless of the values found at the home location. If the zero_flag argument is not set, the data array attribute contains the values found at the home location. If there is a PAGE CONTAINER available in the unused pool, it will be allocated and the value of its name attribute will be returned in the name argument. If, for some reason, a PAGE CONTAINER cannot be allocated (e.g. there are no PAGE CONTAINERs in the unused pool), the operation will fail and return to its caller.

The operation free (name, zero flag) returns the PAGE CONTAINER specified by the name argument to the pool of unused PAGE CONTAINERs. If the PAGE CONTAINER contains all zeroes, the zero_flag argument will be set. Otherwise, the zero_flag argument will be cleared and the contents of the PAGE CONTAINER will be placed at the home location. If the PAGE CONTAINER cannot be freed, the operation will fail.

The read operation can be written as read (name, offset, value). The operation returns, in value, the contents of the data array element specified

by name and offset. Execution of a read operation also sets the used flag attribute of the PAGE CONTAINER.

The operation write (name, offset, value) modifies the contents of the data array element specified by name and offset so that it contains the value given in the value argument. Performing a write operation will also set the used and modified flag attributes of the PAGE CONTAINER.

The five remaining operations return the values of the home attribute and the four flag attributes. They can be illustrated as: get_home (name, home), usedp (name, flag), modifiedp (name, flag), zerop (name, flag), and corep (name, flag). If the PAGE CONTAINER specified in the name argument is currently allocated, these operations will succeed. If not, they will fail.

4.3 Dependencies in the Paging Manager

To satisfy dependency requirements, five object types are available to the paging manager. Three of the types provide storage: the primary memory manager, the paging device manager, and the secondary memory manager. There are also a primitive address space manager and a primitive processor manager. These last two handle the environment and interpreter dependencies of the paging manager, respectively. Component dependencies involve all three storage types. Data arrays stored in primary memory may be referenced immediately, but primary memory can hold only a small number of data arrays at one time. The paging device and secondary memory have larger capacities, but can be referenced relatively slowly because the data arrays must first be copied into primary memory. This means that the paging manager must perform a complicated juggling of data arrays among the available storage areas.

Storage needs for programs and maps are handled by the primary memory manager. This is done for two reasons. First, the programs and maps of the paging manager do not occupy a large amount of storage space. Therefore, the cost of maintaining them in primary memory is small. Second, because the paging manager is heavily used by most of the system, it ought to be as fast and efficient as possible. By storing programs and maps in primary memory, probable frequent references to secondary memory or the paging device for the purpose of accessing programs and maps can be eliminated.

4.4 Discussion

The paging manager provides PAGE CONTAINER objects to many ultimate users. Because of the importance of paging, the manager ought to be simple and efficient. The model, exhibited here, supports exactly one function: referencing pages held in PAGE CONTAINERs; other, more complicated functions (e.g. resource control) are performed by higher layers.

The most striking features of the model are that it is defined entirely in terms of PAGE CONTAINERs (no mention of segments) and that the demand paging nature of the paging manager is hidden. These features are quite appropriate and reasonable. As described, the paging manager provides a useful abstraction of memory for use by higher layers, namely, a set of information containers which can hold pages. Because of storage limitations, there are many more pages than PAGE CONTAINERs. Therefore, users of the paging manager must multiplex the use of PAGE CONTAINERs. This is why PAGE CONTAINERs are A/F objects. They capture the essence of how to manage a scarce, physical resource, i.e. by multiplexing. PAGE CONTAINERs should not be C/D objects because the paging manager would become more complicated and, perhaps, could not even be implemented because of memory shortages.

The paging manager interface should not be expressed in terms of segments because the demand paging abstraction operates completely independent of segmentation. Therefore, it should not know about them. By forcing a segment structure upon demand paging, the paging manager becomes more complex, must deal with considerations which have little to do with its primary function, and loses its generality.

The desire that demand paging be hidden from users is also supported by the independence of demand paging. The internal details of multiplexing logical information containers among physical storage is exactly what the paging manager is supposed to hide. How PAGE CONTAINERs are managed is not important to users of the paging manager.

What is going on when a PAGE CONTAINER is allocated? An allocate operation is prompted by either of two higher layer events. First, some active segment is being assigned a page table in primary memory and its pages need to be accessible. In this case, the contents of the pages are stored in some home, which are in secondary memory. The module performing the activation then calls the paging manager, perhaps indirectly, to assign PAGE CONTAINERs to the non-zero pages of the segment. Second, some page is created for an active segment. Then the page creator needs an empty PAGE CONTAINER to hold the zero page. This is accomplished by assigning a home for the page and calling allocate with the zero_flag set. In section 2.2, a semi-null page was described. In this model, a semi-null page corresponds to an allocated PAGE CONTAINER whose zero_flag is set. Whether semi-null pages are actually stored in their homes is an engineering detail. However, resource control must know if the page is zero in any case.

Since there are a limited number of PAGE CONTAINERs available, an allocate operation could fail because there are none which are free. In this case, the paging manager should not automatically free one. First, the paging manager does not have enough information about the organization of the system to make an intelligent decision about which PAGE CONTAINER should be freed. Second, even if it could decide, the paging manager would have much difficulty

communicating to higher layers which PAGE CONTAINER was freed and why. There are two solutions to this problem. One is to provide enough PAGE CONTAINERs so that the paging manager would never run out. An upper bound on the number needed will be explained in chapter five. The other solution is to implement a PAGE CONTAINER freer which could be invoked when an allocate fails. Its operation would resemble the page removal part of demand paging. Its sophistication would naturally depend on the frequency of its invocation.

A free operation must occur when the page table of an active segment is being moved out of primary memory. Then, the pages of the segment must be moth-balled in a stable state until the segment is needed again. Any zero pages should be put in the semi-null state during the free, so that resource control, the caller of the paging manager, can put them in the null state. Other pages must be placed in their homes for safe-keeping.

The actual demand paging algorithm is hidden inside of the read and write operations. Details of the algorithm are omitted because they are covered quite well by Huber [1976]. With only minor changes, his multiprocess page control can implement the paging manager. For example, Huber discusses the locking issues surrounding the global page table lock and proposes several alternatives.

4.5 Extensions to the Paging Manager

In section 2.2, one of the topics discussed was that the supervisor uses several kinds of special segments for various purposes. Most of these are completely static in length so the supervisor disables the quota mechanism on them. In the context of this model, such segments are not subject to resource control and can be factored out. Therefore, it is appropriate to mention that such segments can be implemented directly on top of the paging manager. This introduces two new type managers, which manage SUPERVISOR SEGMENTs and the SUPERVISOR ENVIRONMENT. They support, in part, the component, map, program, and environment dependencies of higher layers. The SUPERVISOR SEGMENT manager provides paged segments for the exclusive use of the supervisor. These segments are quite different from user segments in that they are not associated with quota cells and rarely, if ever, change lengths. The SUPERVISOR ENVIRONMENT manager controls the naming environment of processes executing in the supervisor. There are two reasons for introducing these new type managers. The first is that the hardware, on which Multics is implemented, prefers to execute in a segmented address space. The segments may or may not be paged. Because of this limitation, the hardware cannot operate in a purely paged manner. Second, paged segments of the type described can be extremely useful to the supervisor. Without them, much of the supervisor would have to permanently reside in primary memory. This, of course, requires the system to include a large primary memory to hold the supervisor. By placing much of the supervisor in paged segments, more primary memory can be devoted to PAGE CON-

TAINERs. How these two managers interact with resource control and segment support will be discussed in chapter five.

4.6 Further Thoughts

A somewhat hidden issue in this model is the interplay between software and hardware. The traditional view is that the hardware is more primitive than the software. However, so far in the model, no mention has been made of the hardware. A very reasonable implementation might be constructed as follows: The read operation, say, is always invoked in hardware. If the data array is in primary memory, the value of the proper word will be returned without ever resorting to software. If the data array is not in primary memory, the hardware will detect this and transfer (fault) to software at the same layer. The software can then copy the data array into primary memory and restart the hardware read. The agent, hardware or software, which performs the operations is immaterial to the type manager and layering constructs. This theme will reappear in chapter five.

4.7 Summary

As seen in chapter two, the principal adverse impact of excess complexity on page control was that the same routine performed both demand paging and the creation and deletion of pages. The model outlined here shows how to perform only the demand paging function. This allows the more complicated create and delete operations to fit into a context more suited to their complex natures.

Chapter Five

Resource Control

Resource control in a virtual memory manager is very tricky. On one hand, page creation and deletion is a frequent occurrence and must be handled efficiently. On the other hand, maintaining the entire file system hierarchy of directories and quota cells in a readily accessible state is simply infeasible because of sheer numbers. Therefore, to perform resource control, given the policy constraints, a subset of the file system hierarchy (those directories and quota cells currently receiving the most usage) should be accessible. This chapter will model the desired behavior of the resource control part of virtual memory management using two type managers.

PAGEMENTs are a new kind of object, in the sense that they do not fit immediately into the jargon and structure of Multics. In structure, PAGEMENTs closely resemble segments. They are, in essence, active segments with page tables. In chapter six, active segments without page tables will be discussed. Since our task is to separate the functions of the virtual memory manager, PAGEMENTs may seem out of place in this layer of the model. They are needed here because of constraints placed on the creation of pages. A page may be created and added to a segment only if three conditions are met: there is quota in the proper quota cell against which the segment is charged, there is space available on the proper physical volume to hold the contents of the page, and there is room in the segment for a new page. Because of the third

condition, which is imposed by the resource control policy, resource control must be aware of the structure of segments.

QUOTA CELLs are introduced as a formal type to hold those elements of the quota cell tree which are currently in use. The remainder of the tree is maintained by the file system. QUOTA CELLs have a similar relationship to quota cells as PAGE CONTAINERs have to pages. To permit reuse, both PAGEMENTs and QUOTA CELLs are A/F objects.

The idea of partitioning memory objects into active and inactive elements because of constraints recurs often. One example is placing some pages in primary memory while the rest are in secondary memory. The constraint is the size of primary memory. Another example is active and inactive segments (see chapter two). The constraint, here, is the amount of virtual memory which can be devoted to AST entries. Janson [1976] discusses the nature and advantages of this idea, as applied to objects and type managers.

5.1 QUOTA CELLs

A QUOTA CELL forms the cost center for the storage accounting system. For accounting purposes, QUOTA CELLs help maintain records of how much storage was used over some period of time by a set of segments. For resource control, the unit of account is one page, but pages are grouped together into cost centers at the QUOTA CELL layer. At this layer, pages are aggregated into PAGEMENTs and entire PAGEMENTs are charged against a single QUOTA CELL. This corresponds to the Multics policy that sets of segments are charged against a single quota cell. Although it is possible to permit the pages of a PAGEMENT to be charged against different QUOTA CELLs, no meaningful use for such generality has yet been found.

QUOTA CELLs are A/F objects, where the number of QUOTA CELLs available is limited by the amount of memory given to the QUOTA CELL manager for storage of components. As with PAGE CONTAINERs, QUOTA CELLs are A/F objects because a single QUOTA CELL can be reused indefinitely to hold different quota cells, as the set of most-used quota cells changes.

5.1.1 QUOTA CELL Attributes

QUOTA CELLs have four attributes: name, frame quota, frames used, and time-frame product. The name of a QUOTA CELL serves to distinguish it from other QUOTA CELLs. The frames used attribute is a non-negative integer which represents the amount of storage (number of frames) currently allocated to segments and PAGEMENTs which are charged against this QUOTA CELL. The frame

quota is a non-negative integer which acts as an upper bound on frames used; the value of frames used may not exceed the value of frame quota. The frame quota and frames used attributes are primarily used for resource control. The time-frame product is used for accounting purposes. The attribute is automatically maintained by the QUOTA CELL manager and is the time integral of the values of frames used since the value of the time-frame product was last reset to zero.

It must be stressed that the initial conditions, when a QUOTA CELL is allocated, are very important. This layer of the virtual memory manager is not sophisticated enough to handle resource control all alone. This layer provides a sort of cache for quota cells already existing in the file system hierarchy [Janson, 1976]. The values in a QUOTA CELL simply reflect the values of the file system quota cell which it holds. These values are based on the status of all segments in the file system and not just the subset of active segments. This is why a QUOTA CELL must hold the quota and frames used for segments which are not even active.

5.1.2 QUOTA CELL Operations

Six operations can be performed on QUOTA CELLs. The first is allocate (quota, used, time-frame_product, name). This selects one QUOTA CELL from the unused pool, sets the values of its frame quota, frames used, and time-frame product attributes to the values of quota, used, and time-frame_product, respectively, and returns the name of the QUOTA CELL. This operation will fail if there are no unused QUOTA CELLs available for allocation.

Conversely, QUOTA CELLs may be freed. The free operation can be written as free (name, quota, used, time-frame product). This returns the specified QUOTA CELL to the unused pool and indicates the final values of its frame quota, frames used, and time-frame product attributes. If the name argument does not refer to an allocated QUOTA CELL, the operation will fail. It is the responsibility of the file system to merge the output values into the file system copy of the quota cell.

The frames used attribute of a QUOTA CELL is the sum of the frames used of all segments and PAGEMENTs which are charged against it. Since segments and PAGEMENTs can grow and shrink in size, an operation is needed to change the value of frames used. This operation is change_used (name, quantity). The change_used operation will fail if name does not refer to an allocated QUOTA CELL or if the result of changing the value of frames used by the value of quantity would be less than zero or greater than frame quota. Most of the changes in frames used occur because of actions by the PAGEMENT manager. However, if an inactive segment is deleted or truncated (shortened), the attributes of the proper quota cell or QUOTA CELL must be updated.

Most of the time, the storage used by segments is not being charged against a QUOTA CELL. Instead, the storage charges are accumulated by the file system at a higher, more static layer. Periodically, the accounting system executes a billing routine which counts the values of the time-frame products, resets them, and prints bills for users. During this process, some segments will be charging against QUOTA CELLs. Clearly, the accounting system must be able to extract these charges from both quota cells and QUOTA CELLs. This is done by providing the operation reset_time-frame_product (name,

product). This retrieves the current value of the proper time-frame product attribute and then resets the value to zero. Storage charges will start to accumulate again at that time and continue until the product is again reset. The operation will fail if the name argument does not represent an allocated QUOTA CELL.

The fifth QUOTA CELL operation allows a user to transfer frame quota from one QUOTA CELL to another. The operation is move_quota (source_name, target_name, quota_quantity). The value of quota_quantity must be a non--negative integer. This operation will decrease the frame quota of the source_name QUOTA CELL by the amount given in quota_quantity and increase the frame quota of the target_name QUOTA CELL by the same amount. The result of this operation must leave the source_name and target_name QUOTA CELLs consistent (i.e. 0 $\leq$ frames used $\leq$ frame quota). The operation will fail if quota_quantity is negative, if the result would leave either QUOTA CELL inconsistent, or if either source_name or target_name does not indicate an allocated QUOTA CELL. This operation is called by the segment support layer described in chapter six.

The final QUOTA CELL operation is a bit complicated. It is move_quota_used (source_name, target_name, quota_quantity, used_quantity). In function, it is quite similar to the move_quota operation, except that it also can transfer amounts of frames used from the source_name QUOTA CELL to the target_name QUOTA CELL. As before, source_name and target_name must refer to allocated QUOTA CELLs and quota_quantity and used_quantity must be non--negative integers. If completion of the operation would leave any QUOTA CELL inconsistent, the operation will fail. This operation is designed to help

change the quota cell against which a segment or PAGEMENT is charged by performing the necessary transfer of frames used. For a change in the QUOTA CELL attribute of a PAGEMENT, this operation is called by the change_QUOTA_CELL operation of the PAGEMENT manager (see section 5.2.2). This operation can also be called by the segment support layer of chapter six.

Why is this operation so complicated? On the surface, it would seem that this operation could be handled by using the simpler move_quota and change_used operations. Consider Figure V-1. Segment DATA is currently being charged against QUOTA CELL 1. The user wants to have DATA charge against QUOTA CELL 2. No sequence of move_quota and change_used operations will

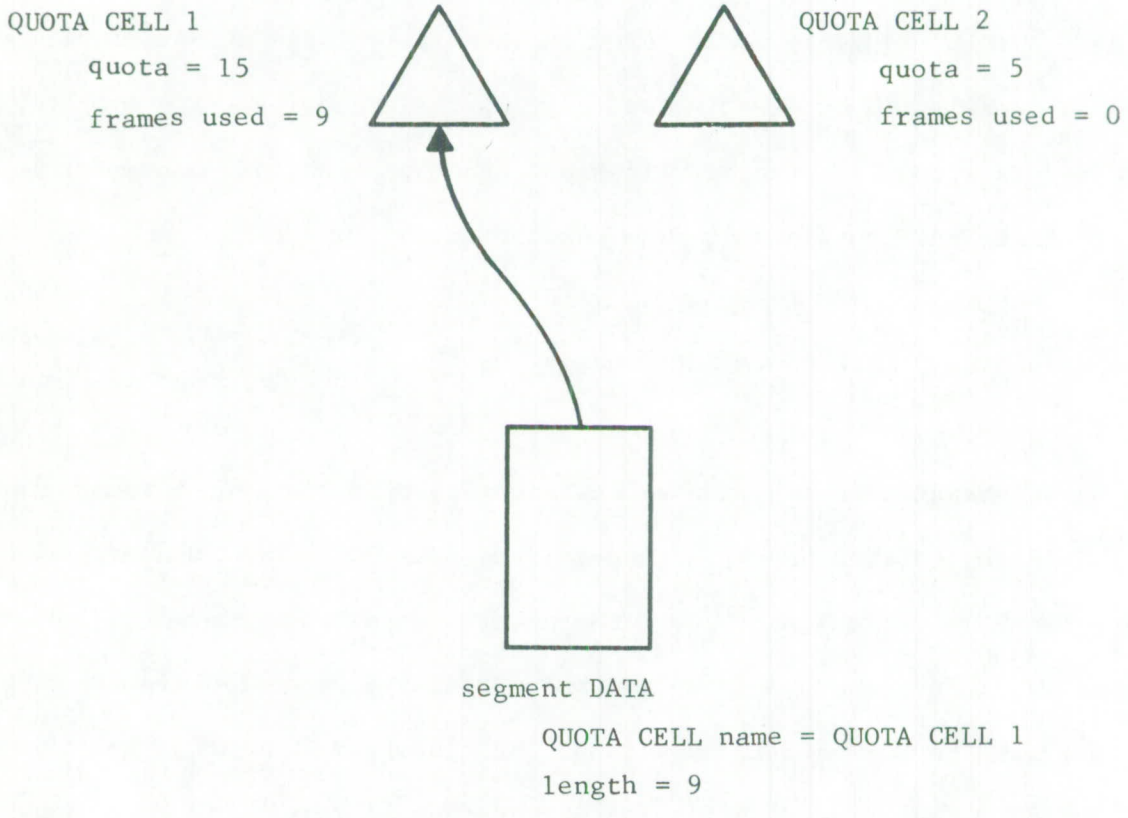


Figure V-1 Moving Quota with Segments

effect the desired change without either forcing one of the QUOTA CELLs to be inconsistent or using a third QUOTA CELL as an intermediate. On the other hand, the operation can be performed by move_quota_used (QUOTA CELL 1, QUOTA CELL 2, 9, 9) and change_QUOTA_CELL (DATA, QUOTA CELL 2) (see section 5.2.2).

The move_quota_used operation is so complicated because QUOTA CELLs are only the bottom piece of the resource control mechanism. The policies of resource control are defined on the quota cell tree in the file system hierarchy. In order to implement the policies in the lower resource control layer, some complication is required. This operation seems the best way to implement the policies and yet minimize complication. Alternatively, the functions performed by move_quota_used could be handled exclusively at some higher layer. This, however, would force a potentially large number of segments and quota cells to be deactivated to allow the operation. The amount of time involved to accomplish the deactivations and the potential delays forced on many processes make such a mechanism unacceptable.

5.1.3 Dependencies in the QUOTA CELL Manager

The paging manager is implemented at a low layer of the system to provide PAGE CONTAINER objects for use by higher layers. The QUOTA CELL manager uses SUPERVISOR SEGMENTs for the storage of components, maps, and programs. SUPERVISOR SEGMENTs, in turn, are made up of PAGE CONTAINERs requested from the paging manager. The QUOTA CELL manager must be given some amount of memory in SUPERVISOR SEGMENTs at system initialization but should rarely, if ever, need more. Those rare occasions would be necessitated by a desire to improve per-

formance, and should be handled by dynamic reconfiguration, rather than allow the QUOTA CELL manager to directly request more SUPERVISOR SEGMENTs. The interpreter needs of the manager can be satisfied by the same low layer processor manager that takes care of the paging manager. Environment dependencies must be met by a more sophisticated type manager than the one which serves the paging manager. The reason for this is simple: the QUOTA CELL manager executes in a SUPERVISOR SEGMENTed environment, whereas the paging manager executes in an unpaged environment. The environment type manager needed is the SUPERVISOR ENVIRONMENT manager which was described briefly in chapter four.

5.2 The PAGEMENT Manager

In Multics terms, PAGEMENTs represent active segments which are currently being referenced. They provide the illusion of large, linear, variable-length arrays of words, rather than sets of fixed-length PAGE CONTAINERs. PAGE CONTAINERs are allocated from the paging manager to PAGEMENTs and are ordered into a linear array. This array allows the conversion of a reference to a word in a PAGEMENT into a reference to a word in a PAGE CONTAINER.

The PAGEMENT abstraction is necessary in the resource control layer because of three of the policy constraints in page creation: pages may be created for a segment only if there is room in the segment for them, all pages of a segment must reside permanently on the same physical volume, and all pages of a segment must be charged against the same quota cell. Given simpler policies, PAGEMENTs would not be needed in resource control.

5.2.1 PAGEMENT Attributes

PAGEMENTs are somewhat complex objects and have twelve attributes. They are: name, size, length, frames used, QUOTA CELL name, used flag, modified flag, physical volume, core count, page table, page table modified flag, and data array. PAGEMENTs can be allocated only in the discrete sizes of 4, 16, 64, or 256. The size refers to the maximum number of PAGE CONTAINERs that can be elements of the PAGEMENT (i.e. the number of PTW's in the page table). If a user wants to grow a PAGEMENT beyond this size, a new PAGEMENT must be allocated. Frames used indicates the number of PAGE CONTAINERs which have been

allocated to the PAGEMENT. The length corresponds to segment length, discussed in chapter two.

The PAGE CONTAINERs used by PAGEMENTs are charged against a QUOTA CELL. The QUOTA CELL name attribute indicates against which QUOTA CELL the PAGEMENT is being charged. This attribute thus indicates which QUOTA CELL's frames used attribute must be changed when the PAGEMENT grows or shrinks. As with PAGE CONTAINERs, the used and modified flags tell whether the PAGEMENT has been used or modified since the PAGEMENT was allocated or since the flag has last been tested. All of the pages in a PAGEMENT must have homes on the same physical volume. The physical volume attribute holds the name of that physical volume. If the PAGEMENT grows or shrinks, homes must be allocated or freed from that physical volume. The core count indicates the number of pages of the segment which are currently in primary memory. It is used by the segment deactivation algorithm. It is calculated by counting the number of PAGE CONTAINERs whose core flags are set.

A page table is essentially the map of a PAGEMENT. It indicates which PAGE CONTAINERs hold the information in the PAGEMENT's data array. Entries in the page table may either be page homes or special null values. A null value indicates that the corresponding information in the data array is all zeroes and, thus, needs no PAGE CONTAINER to store it. The page table modified flag indicates whether the page table attribute has changed since the PAGEMENT was allocated or since the flag was last tested. The last attribute, the data array, holds information. It appears as a linear array of words, any of which may be referenced whenever the PAGEMENT is allocated.

The page table is central to the PAGEMENT concept. It provides enough structure to groups of PAGE CONTAINERs to support a variety of sophisticated resource control policies. It also fits smoothly into the existing Multics view of a segment. Finally, separating page tables from active segments is an excellent way to minimize functional entanglement in both resource control and segment support. In resource control, we wish to capture a minimal segment structure on which to build. In segment support, as we shall see in chapter six, several functions interact in complex ways. By providing page tables in resource control, the complexity of segment support can be reduced.

5.2.2 PAGEMENT Operations

Since a PAGEMENT is a more complicated object than a PAGE CONTAINER, PAGEMENT operations are also more complicated. The first is the allocate operation. It can be written as allocate (size, physical_volume, page_table, quota cell, name). The size argument tells which of the four different sizes of PAGEMENTs should be allocated. There are four unused pools, one for each size, and the operation will select a PAGEMENT from the appropriate pool. The other three input arguments initialize the values of some of the attributes. The values of the other attributes can be derived from them. For example, the starting values of the three flag attributes is "false"; the values of length and frames used can be determined from the page table. The information in the data array is defined to be the information stored in the respective homes listed in the page table. This operation will fail if there are no free

PAGEMENTs of the proper size. In Janson's software cache structure, the allocate operation activates or encaches a segment data abstraction as a PAGEMENT.

Conversely, the operation free (name, physical volume, page table, quota cell, used flag, modified flag) frees the PAGEMENT specified by name. The final values of the physical volume, page table, QUOTA CELL name, used flag, and modified flag attributes are returned by the operation. Naturally, the operation will fail if name does not refer to an allocated PAGEMENT. This operation corresponds to deactivating or decaching a segment data abstraction in Janson's structure.

It is important to stress, here, that the attributes of a PAGEMENT, as visible to a PAGEMENT user, are different from their internal representations. In this case, the visible elements of a page table are page homes or null values. Internally, however, the elements are PAGE CONTAINER names or null values. In other words, within the PAGEMENT manager, page homes are represented as PAGE CONTAINER names. The homes are passed on to the paging manager. Similarly, inside the paging manager, semi-null pages are hidden. When a PAGEMENT is allocated, the PAGEMENT manager, in turn, allocates PAGE CONTAINERs for those pages which have homes. When the PAGEMENT is freed, the PAGE CONTAINERs are also freed and the list of homes returned to the caller.

These transformations are important because the different states allow a segment to be represented with different dynamic capabilities. In its most static state, a segment resides only in secondary memory, and its page table is in its VTOCE, which is also in secondary memory. Almost nothing can change a segment in this state, but if the system crashes, the segment is very likely to survive the crash. In chapter six, the next state will be presented. The

third state occurs when the pages of the segment are in primary or secondary memory and the page table is in primary memory. This state is represented by PAGEMENTs. Non-null pages are held in PAGE CONTAINERs and are very dynamic. They can move from secondary memory to primary memory and back again quickly. Null pages have no homes and, if referenced, must be created. In the third state, a segment is more likely to be damaged by a system crash. By dividing the dynamics of a segment into states, different, useful information containers can be provided which abstract the important features in different layers. Also, low level implementation details can be hidden from higher layers. The file system does not need to know and cannot be helped by knowing the mechanics of moving PAGE CONTAINERs from secondary memory to primary memory.

The most frequent PAGEMENT operations are read (name, offset, value) and write (name, offset, value). Both operations translate the name and offset into a PAGE CONTAINER name and offset and call on the paging manager to obtain or modify the value. If the page referenced by offset is a null page, the read operation will automatically return the value zero without calling the paging manager. The write operation, in this case, will allocate a PAGE CONTAINER. This is done by allocating a home on the proper physical volume, incrementing the frames used attribute of the QUOTA CELL specified in the QUOTA CELL name attribute of the PAGEMENT, calling the allocate operation of the paging manager, changing the frames used, page table, and page table modified attributes of the PAGEMENT, and, if necessary, changing the length of the PAGEMENT. Then the write is completed. If no home can be allocated, the write will fail. Performing either operation will cause the used flag attribute to be set; performing a write will set the modified flag. Either opera-

tion will fail if name does not refer to an allocated PAGEMENT or if the offset is not within the data array of the PAGEMENT.

The write operation contains the essence of the resource control mechanism. When triggered by a quota page fault (see section 2.2), the write operation automatically creates pages. Other layers become involved only if, for some reason, a page cannot be created. This is an elegant method for resource control. Unless a page cannot be created, resource control operates quietly and smoothly, but in a well-defined manner.

Three predicate operations exist to return the values of the flag attributes. They are: usedp (name, flag), modifiedp (name, flag), and page_table_modifiedp (name, flag). If name is the name of an allocated PAGEMENT, the operations will indicate the values of the used flag, the modified flag, or the page table modified flag, respectively. Otherwise, the operations will fail. If the flag is set, the operation will return "true" and clear the flag.

As an aid to reliability, the get_page_table (name, physical volume, page table) operation is provided. If the name argument represents an allocated PAGEMENT, the operation will return the values of the physical volume and page table (list of homes) attributes. This can be used, for example, as follows: Periodically, a higher layer manager can poll the PAGEMENT manager to see if the page tables of any of the PAGEMENTs have changed. If any page tables have changed, the higher layer manager can extract them and store them in a safer and more reliable place (e.g. in their VTOCE's). The values of the length and frames used attributes can be inferred from the page table.

Two operations allow for the manipulation of the QUOTA CELL name attribute. They are get_QUOTA_CELL (name, cell name) and change_QUOTA_CELL (name, new_cell_name). The first operation simply returns the current value of the QUOTA CELL name attribute. The second changes the attribute's value, for reasons discussed in section one of this chapter. As usual, these operations will fail if name does not refer to an allocated PAGEMENT. To preserve the consistency of QUOTA CELLs, the change_QUOTA_CELL operation calls the move_quota_used operation of the QUOTA CELL manager (see section 5.1.2). The call is made to transfer the proper amount of frames used from the old QUOTA CELL to the new QUOTA CELL. Note that during a change QUOTA CELL operation, the PAGEMENT manager must inhibit any operation on the specified PAGEMENT which would change its length. Otherwise, the value of frames used might be inaccurate.

For the purposes of this model, change_QUOTA_CELL will always call move_quota_used with the values of quota_quantity and used_quantity set to the value of frames used of the PAGEMENT whose attribute is being changed. In any case, the value of used_quantity must equal frames used. However, if the target QUOTA CELL has sufficient unused quota, the value of quota_quantity could be less. Operations could be introduced here which would take advantage of the unused quota.

For the convenience of the segment deactivation algorithm, the get_core_count (name, count) operation is provided. Given the name of an allocated PAGEMENT, it will examine the values of the core flag attributes of component PAGE CONTAINERs. The operation will return the number of core flags which are set.

The truncate (name, length) operation provides a relatively efficient method of discarding unnecessary pages. If the length of the PAGEMENT is greater than the value of the length argument, the PAGEMENT will be shortened by freeing PAGE CONTAINERs off of the end of the PAGEMENT. If the length is less than or equal to the value of the length argument, no change will occur. This operation is equivalent to writing zeroes into the relevant PAGE CONTAINERs, but is less time-consuming.

The final PAGEMENT operation is also motivated by efficiency considerations. The operation is move_contents (name, size, new name). The value of the size argument must be greater than the current value of the size attribute. This operation says to allocate a larger PAGEMENT, of the size specified, move the contents of the PAGEMENT given by name into the new one, free the old PAGEMENT, and return the name of the new one. This is equivalent to freeing the old PAGEMENT and allocating a larger one for the data array. This operation is faster than freeing and reallocating because the component PAGE CONTAINERs do not have to be freed.

Move_contents is an optimization towards quickly growing segments. The corresponding operation in the current system has proven effective because segments grow frequently. A segment grows because some process references a page outside of the PAGEMENT length. Usually, the page can be created immediately and the reference restarted. Sometimes, the page is also outside of the PAGEMENT size, which requires a larger PAGEMENT. The move_contents operation speeds up the allocation of a larger PAGEMENT for this purpose.

5.2.3 Dependencies in the PAGEMENT Manager

The dependencies of the PAGEMENT manager are mostly the same as the dependencies of the QUOTA CELL manager. SUPERVISOR SEGMENTs are used for the storage of components, maps, and programs; the low layer processor manager provides processor resources; and the SUPERVISOR ENVIRONMENT manager structures the naming environment. The difference is that the PAGEMENT manager also depends on the QUOTA CELL manager through the change_used and move_quota_used operations.

Because of hardware restrictions, the page table of a PAGEMENT must be in primary memory. Therefore, page tables are kept in an unpaged segment, like those used to implement the paging manager. All other PAGEMENT components can be stored in SUPERVISOR SEGMENTs.

5.3 How PAGEMENTs and QUOTA CELLs Fit Together

The PAGEMENT and QUOTA CELL managers must cooperate closely to provide the middle layer of the virtual memory manager. Although each supports one type, the abstraction desired is produced by a fusion of the two.

Two obvious connections between the managers are that the PAGEMENT manager calls the change_used and move_quota_used operations of the QUOTA CELL manager. These calls constitute dependencies of the PAGEMENT manager on the QUOTA CELL manager.

A more important connection has to do with initial conditions. If the managers are given correct data on which to operate, one can be convinced that the correctness of the data will be preserved. If, however, a higher layer, e.g. the segment support, passes faulty or maliciously contrived data, the results at the higher layer are unpredictable. The burden of consistency, here, rests on the higher layer manager.

5.4 Resource Control and PAGE CONTAINERs

Two final items must be discussed. Both the PAGEMENT manager and the SUPERVISOR SEGMENT manager use PAGE CONTAINERs to implement different kinds of segments: the PAGEMENT manager uses them for file system segments available to users, and the SUPERVISOR SEGMENT manager uses them for SUPERVISOR SEGMENTs. The proper way to view this is that PAGE CONTAINERs are partitioned between the two managers. How do we avoid confusion about which PAGE CONTAINER is allocated for which manager? One way is to rely on a careful implementation so that neither manager tries to inadvertently or maliciously use the wrong PAGE CONTAINER. This solution will work, but places a larger burden on certification procedures to guarantee correctness. A more sophisticated solution would involve tagging each allocated PAGE CONTAINER with the name of the manager for which it is allocated. Then, the paging manager could check each operation to make sure that only the proper manager is performing it. In Multics, this solution can only prevent inadvertent use of the wrong PAGE CONTAINER because there are no enforced protection barriers in this layer of the system. Any malicious program can subvert such consistency checks. In a more advanced architecture, perhaps using domains, a checking mechanism could prevent all incorrect uses.

A perceptive reader may have noticed that the model has departed from the Multics system because another layer of indirection has been added to the addressing mechanism. In Multics, references go directly from the SDW to the PTW to the proper word. In the model, they go from the SDW to the PTW to the paging manager to the word. The difference is that in Multics, the physical

location of the page is kept in the PTW. In the model, the physical location is hidden within the paging manager. This was done to allow the disentanglement of the resource control and demand paging functions. By introducing an extra translation layer, we can see much more clearly what is going on inside of the virtual memory manager.

In Multics, when a segment is activated, a page table is allocated and a set of PAGE CONTAINERs is allocated at the same time. However, the size of the set depends on how many PTW's in the page table are null. Therefore, the set of PAGE CONTAINERs on which demand paging may operate can change arbitrarily and without any explicit notification of the paging manager. To handle such arbitrary changes, the Multics page control becomes quite complicated because it must frequently check whether its set of PAGE CONTAINERs has been changed. Using the model, we can see that the Multics PTW serves two different purposes. The different purposes are masked because much of the resource control function is performed by page control. In the model, the difference becomes clear. For resource control purposes, the PTW represents the name of a PAGE CONTAINER or a null page. From this, resource control can determine the values of frames used for PAGEMENTs and QUOTA CELLs. For demand paging, the PTW holds the physical location of the page so that demand paging can determine where pages are. Hopefully, a future system designer will realize that in the virtual memory manager, an engineering decision must be made for either addressing efficiency or clarity of structure. The inherent complexity of the system is strongly affected by his choice. The dual function of the PTW again points out the independence of the type manager approach from the boundary between hardware and software.

In section 4.4, discussion touched on the issue of how many PAGE CONTAINERs there should be. Given the fact that the PAGEMENT manager can never allocate more PAGE CONTAINERs than it has entries in page tables, an upper bound on the number of PAGE CONTAINERs needed for the system can be computed. That upper bound is the number of entries in page tables plus a number dependent on SUPERVISOR SEGMENTs. Since the storage requirements of SUPERVISOR SEGMENTs are quite static, one can determine the number of PAGE CONTAINERs needed for them by counting.

5.5 Summary

To support sophisticated policies, the resource control layer of the virtual memory manager cannot operate in a vacuum. It must embody enough knowledge about virtual memory to implement the policies, but should avoid excessive complexity. For Multics, we have designed the resource control layer to be the cache of a software cache structure. Thus, we can maintain the most--needed elements of the file system hierarchy in a readily accessible state, and keep track of required information in a natural form. The internal consistency of resource control will take care of itself if no higher layer attempts to subvert it. However, the consistency of the system as a whole depends on higher layers feeding the proper information to resource control at the proper times.

Chapter Six

Segment Support

The segment support function of the virtual memory manager groups pages together to provide the concept of active segments. Because of the Multics resource control policies, some of this work is already accomplished through PAGEMENTs. However, PAGEMENTs are not active segments, so further extension is necessary.

In contrast to chapters four and five, we will not attempt to model segment support. The reason is that a model would concentrate on the interface between segment support and higher layers. Since the higher layers have not been analyzed and dissected to the same degree as the virtual memory manager, a presentation of the interface would not reveal much about the proper workings of segment support. Also, the reader would get bogged down in technical details for which we have not presented the proper context. Instead, this chapter will be a general discussion of active segments and how to supervise the resource control layer.

6.1 Active Segments

Active segments are best thought of as an encached form of file system segments. They are the first step in allowing words of memory to be referenced. Because there can exist a very large number of segments in the file system, it would not be feasible to maintain a single, unified data base large enough to keep track of them all. In addition, since so much would depend on the correctness of such a data base, one crash could irreparably damage the system. Therefore, necessary information on segments is distributed among several fragmented data bases, each of which is manageably small and relatively safe from the effects of a crash. When a segment is activated, this information is copied into the AST. The information is not deleted from the other data bases, so, in the event of a crash, the system can usually recover without the loss of any information.

In the current system, PAGEMENTs do not exist. Their function is subsumed by active segments. Therefore, it is not immediately clear how to relate PAGEMENTs with active segments. The first order of business in this section is to discuss the nature of the information in the AST. Then we shall see how the information can be divided among PAGEMENTs, QUOTA CELLs, and active segments. Finally, we will discuss why, independent of the resource control policies, the PAGEMENT concept is valid, and simplifies the task of segment support.

6.1.1 Information in the AST

In each ASTE, the information contained consists of five types. The first has to do with the internal management of the AST (thread pointers, allocated flags, and the like). This information depends only on specific management strategies and need not concern us here. The second kind of information determines the segment's context in the file system. Included in this category are: a pointer to the ASTE of the segment's immediate parent (remember, all parent directories of an active segment are active); a chain pointer to an active brother's ASTE; a pointer to the ASTE of a son, if any son is active; and an indication whether the ASTE represents a simple segment, a quota directory, or an intermediate directory. Thus, the relative position of the segment in the hierarchy is maintained. This information is used to help direct many operations of higher layers. For example, it can be used when moving quota from a directory to its son. The operation originates from a call to the supervisor by a user. Using this information, the fields of the proper ASTE's can be modified.

The third kind of information is the quota cell itself. If a quota directory is active, its quota cell is kept in its ASTE. This is used to implement the resource control policies and accumulate storage charges. Unfortunately, the space for a quota cell exists in every ASTE, to simplify AST management. Thus, since few ASTE's represent quota directories, most of this space is wasted.

Fourth, specific information about the segment is kept. This includes the page table, the various length parameters, used and modified flags, and

the number of pages of the segment which are in primary memory. As described in chapters two and five, this is used for address translation, resource control, and demand paging. This is precisely the information which we have placed inside of PAGEMENTs and PAGE CONTAINERs.

The last kind of information addresses the problem that a segment may have a different segment number in different processes (see section 2.2). When a segment is deactivated, the segment fault flag must be set in all SDW's which are connected to the segment (i.e. all SDW's must be disconnected). To do this efficiently, the system needs a list of the SDW's connected to each active segment. The list is maintained by the AST manager (segment control).

6.1.2 Splitting Up the AST

The AST is the primary data base for the virtual memory manager. The information which it contains is used for all of the virtual memory functions which we have described. However, we have carefully tried to disentangle the functions so that they are clearer and easier to understand. If this clarity could not be extended to the AST, our model would be suspect. Fortunately, this is not the case. The information can be neatly divided. The third kind of information becomes the QUOTA CELL. A QUOTA CELL is allocated whenever a quota directory is activated. Conversely, a QUOTA CELL is freed whenever a quota directory is deactivated. This may seem to be parallel to the current system. It is, in the important sense that the QUOTA CELL is available when the directory is active. However, this scheme has the advantages that it fits smoothly into the model and it does not waste as much space in unused QUOTA

CELLs. Further, by providing separate data bases for resource control and segment support, we can feel confident that the two functions do not interfere with each other or permit communication through some data base.

The fourth kind of information becomes PAGEMENTs and PAGE CONTAINERs. A PAGEMENT is allocated whenever a segment is activated. However, a PAGEMENT may be freed before the segment is deactivated. This can be used, for example, to allow intermediate directories to be active, yet not tie up valuable page table resources. This makes PAGEMENTs more usable because the process of freeing one does not require the deactivation of a segment.

The remainder of the information stays within the AST. The first kind obviously belongs there because it is concerned with AST management. The fifth kind must stay in the AST because segments may be disconnected for several reasons, only one of which is for deactivation (see section 2.2). The second kind shows what the real nature of segment support is. Segment support directs the operation of the virtual memory manager. Within it is the information needed to accurately decipher supervisor and user commands which are necessary for operation of the system. In certain respects, the segment support layer acts like a traffic cop: Operations may be received at any time. Segment support must coordinate the execution of the operations to maintain the consistency of the system. This may seem unusual since we have already stated that the role of segment support is to provide support for segments. In fact, the two functions are the same. The only difference is in perspective. Segment support is like a traffic cop as seen by lower layers. Higher layers need know nothing of this. They only see that segment support performs many operations related to virtual memory.

6.1.3 Active Segments and PAGEMENTs

We have stated that PAGEMENTs are included in the resource control layer because of the policies involved. This, however, is not the reason for their existence. PAGEMENTs occupy an important place independent of the resource control policies, although the policies influence their structure. To operate properly, the virtual memory manager requires different kinds of information which have different lifetimes. PAGE CONTAINERs have the shortest lifetimes. One is needed only as long as a page is being referenced. At the other end of the spectrum, active segments have much longer lifetimes. They are needed to make all pages of some segments accessible. They also hold directories which are parents of other active segments. A particular ASTE may only be distantly related to any segment being referenced. In between, there is a need for some information container to hold single segments. The container should allow an entire segment to be referenced, but does not need the hierarchy context of an ASTE. That container is the PAGEMENT. If, as in the current system, this information container is fused with the AST, space is wasted, because the AST must have entries for segments not being referenced. If, instead, PAGEMENTs and active segments are separated, the different functions that they help implement become clearer. The PAGEMENT manager is concerned with ordering pages into segments and allowing them to be referenced. Active segments are used for interpreting operations according to the file system hierarchy.

6.2 Functions of Segment Support

Segment support is entrusted with the responsibility of maintaining the consistency of the virtual memory manager. By this we mean that segment support must supply resource control with the correct data and must return the proper information to the file system. QUOTA CELLs and PAGEMENTs are allocated and freed at the exclusive direction of segment support. The caveats noted in chapter five should therefore be applied to segment support.

Several specific operations, to which previous chapters have alluded, occur within the domain of segment support. First, segment support must allocate and free PAGEMENTs. One way to do this would be to have available as many PAGEMENTs as there are ASTE's and to equate segment activation with PAGEMENT allocation and segment deactivation with the freeing of PAGEMENTs. While conceptually simple, this scheme requires many PAGEMENTs that would not often be used. A better scheme involves fewer PAGEMENTs. A PAGEMENT would be allocated whenever a segment were activated or when an active segment, which does not correspond to a PAGEMENT, were referenced by a user. A PAGEMENT would be freed if one were needed for some other allocation or if a segment which does correspond to a PAGEMENT were deactivated. Naturally, when freeing a PAGEMENT, segment support should try to free that one which has been least recently used. Such a scheme parallels the algorithms used for page replacement in primary memory and for segment deactivation. The parallel should not be surprising, since it is a common one for the management of a scarce resource.

A second operation is the allocation and freeing of QUOTA CELLs. Again, this could be simply accomplished by having as many QUOTA CELLs as ASTE's. Of

course, this, too, wastes space. A recent survey of the MIT Multics system shows that there are about 500 quota directories, as compared with about 1100 ASTE's. A somewhat better scheme, then, would need only as many QUOTA CELLs as there are quota directories. This would save space, but what would happen if a user created more quota directories? A more ambitious algorithm would have even fewer QUOTA CELLs. If segment support needs to allocate a QUOTA CELL and there are none left free, segment support could deactivate a quota directory and its associated subtree, thus freeing a QUOTA CELL. The frequency of forced deactivations of quota directories is strongly affected by the total number of QUOTA CELLs available, so the system should be tuned carefully to minimize it. One way for the system to tune itself would be to have it monitor the forced deactivations of quota directories. If they occurred too frequently, segment support could ask the QUOTA CELL manager to perform a dynamic reconfiguration to obtain space for more QUOTA CELLs. This, of course, would require that a new operation be added to the QUOTA CELL manager.

As part of AST management, segment support must provide to higher layers the facilities to activate or deactivate segments and to add or remove entries from the connected segment list. Higher layers control these operation according to the rate at which segments are referenced.

Segment support handles bound faults. As explained in section 2.2, a bound fault occurs if a user tries to reference a page of a segment which is outside of the bounds of the page table. If the page is within the segment's maximum length, segment support simply calls the move_contents operation of the PAGEMENT manager. Otherwise, an appropriate error message is relayed to the user.

Another situation which must be handled by segment support occurs if the PAGEMENT manager tries to create a page and there is no space left on the proper physical volume for another page home. The PAGEMENT manager informs segment support of the problem. Segment support tries to relocate the segment on another physical volume in the same logical volume, because of the reliability constraint that all pages of a segment must have homes on the same physical volume. Segment support must try to find another physical volume and move the segment to it. If this cannot be done, the user must be informed that he cannot further grow the segment.

The above operations usually occur as the result of a segment, bound, or page fault. A different class of operations handled by segment support occur because of user calls to the supervisor. It includes such operations as deletion of segments, creation and deletion of quota cells, movement of quota among quota cells, and changes to segment attributes. The principal action of segment support is to interpret these operations according to the directory hierarchy pointers in the AST and invoke the proper operations of the resource control layer.

For example, suppose that a user wants to change the maximum length of a segment to X. The maximum length is maintained and enforced by segment support, when the segment is active. However, segment support will also check with the PAGEMENT manager, if the segment is associated with a PAGEMENT, to make sure that the length of the segment is not already greater than X. If it is, the operation cannot be completed.

6.3 Summary

Segment support is the guiding light of the virtual memory manager. It must coordinate activities and maintain the context of active segments in the file system. The operations for which it is responsible fall into two classes: those arising from hardware faults during address translation, and those resulting from user and supervisor software calls. The information currently in the AST can be neatly divided among the resource control and segment support layers of our model.

Chapter Seven

Conclusion

This thesis has attempted to model virtual memory management in a computer system. As part of the modeling effort, the methodologies of type extension and layers of abstraction were used extensively. Type extension and layering have a broad applicability to computer systems. Several new languages use these ideas as a basis for the structuring of data. In this thesis, we have attempted to show their usefulness in operating systems. The resulting model and specification is, at least on paper, simpler and easier to understand. In the future, we can look forward to hardware support for objects. Therefore, it is important now to develop the necessary tools to use them for the construction of operating systems.

The significance of this thesis is more than that of a simple paper design. The system modeled is not a toy. Multics is a large, complex operating system sold commercially by Honeywell. The use of a real system is important to demonstrate that the issues involved are not only academic. The basic issue is simplicity. The task of proving the correctness of a system, either formally or informally, is much easier on a simple system than on a complex one. Perhaps more important, an operating system is maintained by people. Over time, the system evolves and the set of people who maintain it changes. The maintainers must understand how and why the system works. Because of this, a simpler system is easier to maintain than a complex system.

7.1 Results

Chapter two discussed some of the problems found in the current Multics virtual memory manager. Although the details are quite specific to this implementation, the general problem of functional entanglement is a common phenomenon of large software projects. In chapter three, we considered techniques which control complexity. The keys were modular and hierarchical structures. Then, we argued how layering and type extension can help achieve a modular hierarchy.

A specific model of demand paging and resource control was presented in chapters four and five. This model is expressed in terms of objects. The point of the model is that the underlying functionality of the virtual memory manager can be preserved, but can be implemented in a simpler, more structured way. Although designed for Multics, the model structure is applicable to any implementation of virtual memory because the basic problems remain the same.

In chapter six, we discussed the final layer of the model, segment support, in general terms. The reason for this more general presentation is that there is functional entanglement among segment support and higher layers of the supervisor. A simple model of segment support cannot be designed without a detailed analysis and redesign of the higher layers, which was beyond the scope of this study. One of the important points in the thesis is the impact of module dependencies on the design of a system. The hierarchy of modules in any system is defined by those dependencies.

The structure of the final model is given in figure VII-1. The circles represent type managers and the arrows are dependencies. The horizontal lines

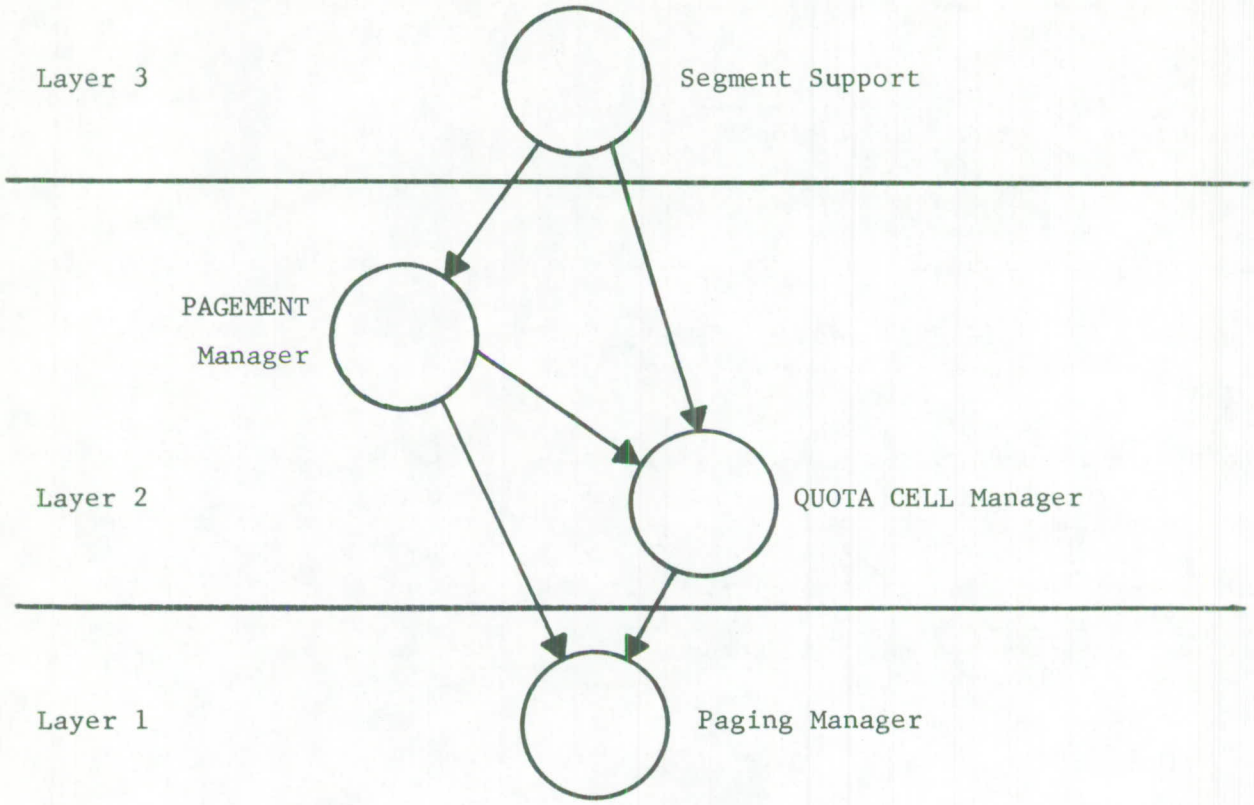


Figure VII-1 Final Structure of the Model

show the layering. For clarity, dependencies have been omitted on managers such as the SUPERVISOR SEGMENT manager. Although some of the dependencies have been omitted, the entire dependency graph can be drawn, and the graph conforms to the layering structure given in the figure. Note that in the given structure, the QUOTA CELL manager cannot be placed in a separate layer, because then there would be a dependency of the segment support layer on the QUOTA CELL manager which would by-pass the PAGEMENT manager.

7.2 Differences between Multics and the Model

A new model or implementation can differ from an existing one in three ways. The first is that the external functionality of the new model or implementation is visibly different from that of the original. An experienced user can notice the differences, so the propriety of the functional changes must be examined. Second, the internal structure and functions can be rearranged without visible change. Such a difference would be visible only if the model or implementation were examined in detail. Third, and most subtly, the new model or implementation can represent a different perception of the functions that need not force any physical structural modifications, i.e. a logical restructuring.

7.2.1 Visible Functional Differences

The model presented here departs from the visible functionality of Multics in two ways. The first difference is that in the model, reading a null page will not invoke the page creation mechanism. This change is visible to users through quota cells and through record quota overflow faults. Chapter two discussed the inconsistencies of creating a page when reading. The point is that a read should not modify the state of any segment or quota cell. This change is not due to any feature of the model and is not original here; it remedies a basic flaw in the design of Multics. This difference does require a change to the hardware, but several acceptable hardware modifications are known. While philosophically important, this difference can be

omitted from the model or from an implementation. A more detailed discussion of the issues surrounding the reading of null pages is given by Stern [1976].

Second, the model modifies the external appearance of intermediate directories. In Multics, a count of frames used is maintained for all directories, whether or not they are quota directories. For quota directories, this count is identical to the count in the model, namely, the number of frames charged to the quota cell. For an intermediate directory, the count is identical to what it would be, if the directory were a quota directory. Therefore, in Multics, all directories have some sort of a quota cell, but in some directories, the quota element of the quota cell is ignored. This can be done, in the current Multics, because page control must step through all intermediate directories when finding a quota cell. Thus, incremental changes to all quota cells affected by creation or deletion of a page are very easy. The count, however, has little meaning for intermediate directories. In the model, we have omitted frames used from intermediate directories so that the resource control layer may immediately access the proper quota cell without referencing any directories. In this way, the potential dependence of resource control on directories is eliminated. This difference is also visible to users because, in the model, frames used is not maintained in intermediate directories. The advantages, however, are that referencing a quota cell is faster and the structure of resource control is simpler because, in every case, the quota cell may be accessed directly.

7.2.2 Invisible Structural Differences

The internal structure of the virtual memory manager model also departs from the structure of Multics. The first difference of this sort is related to the second one noted in section 7.2.1 because it also involves quota. In Multics, all changes to the quota cell hierarchy are accomplished through the move_quota operation. This operation transfers quota from one quota cell to another and creates or deletes quota cells where appropriate. It is facilitated by requiring that quota cells be available for all directories superior to any active segment, but it also requires that all segments in one directory be charged to the same quota cell. Using the move_quota_used operation of the model, the same interface can be presented to users. This can be done as follows: If both source and target directories are quota directories, only the amount of quota in each quota cell must be changed. If, however, the target directory does not have a quota cell, a quota cell must be created for it, and all segments in the subtree under the target directory which were charged against the source directory's quota cell must be changed to charge to the new quota cell. This change is performed by invoking the change_QUOTA_CELL and move_quota_used operations on each affected segment. Since the process of moving quota is infrequent, any loss of efficiency will be insignificant. Besides eliminating dependence on directories, the reason for implementing changes to the quota hierarchy as in the model is to allow flexibility in the resource control policy. This point will be expanded in section 7.3. Note that this difference and the second one above occur jointly. Accept both or

neither. They are introduced to simplify the model and to clean up the quota cell concept.

The second structural difference is that the model has three principal modules, while the current Multics system has two. This difference has an important implication: The management of logical changes to segments (i.e. the creation and deletion of pages) has entirely different concerns than the physical management of pages. The two need not and should not be intertwined. Therefore, we can consider the usefulness of physical management (paging) independent of logical management (resource control). The advantage for the implementation is that the problems mentioned in section 2.3 and other similar problems have been solved. Several supervisor programs and data bases, most notably the FSDCT, can be paged in a natural way and resource control can be performed in a modular fashion. This results in economies of mechanism and a cleaner structure. The new structure facilitates understanding and comparison of the implementation with the specification. Examining the model, we can more easily appreciate how virtual memory works.

The final difference in structure, as noted in chapter four, is the added level of indirection in the segment addressing path. To implement the model fully would require a completely redesigned CPU, which is not feasible for Multics. The real point in this difference is not to suggest a hardware change at this time. Instead, we want to alert system designers that the Multics PTW serves two different purposes for two different layers of the virtual memory manager. This presents a trade-off between clean structure and efficiency which should be recognized. Given the trend in hardware speeds and prices, this trade-off may soon swing towards cleaner structure.

7.2.3 Philosophical Differences

There is a qualitative difference between the model and the Multics of today that is quite general to the study of operating systems. This difference exists independent of whether any code is rewritten. It is the strict view, in the model, of the system as a hierarchical collection of extended types, as opposed to the Multics view which organizes the system loosely. The most obvious use of a strict object approach is in system verification. Verification, of necessity, requires rigor, and thus cannot be applied to any loosely defined system. To this end, objects and types are quite appropriate.

Type extension represents more than a verification tool. It is also a general method of describing systems. It is a way of looking at a system which neatly captures the fundamentals of the system's dynamics. Type extension is concise, precise, and very flexible. Thus, it is amenable to formal description and manipulation. This means that type extension is a more powerful method of system description than simply describing a system in terms of modules.

7.3 Resource Control

In this thesis, we have examined some of the issues of storage resource control and how they relate to virtual memory. In this section, we will attempt to sum up those relationships. There are many policies which can be used to control the usage of storage resources. They range from almost non--existent, where the only constraint is the amount of physical storage available to the system, to sophisticated, like those in Multics. This variety is one of the reasons for splitting resource control and demand paging. Any virtual memory implementation will need a standard demand paging algorithm to support it. The resource control policies, however, are part of the virtual memory definition, and will change as the virtual memory implementation changes.

In general, resource control involves four elements: a policy maker, a mechanism, a set of requestors, and a supplier. The policy maker dictates the policies under which all allocation and freeing decisions are made. The mechanism enforces the policies. It must be at the same layer or lower than the policy maker. Of course, there must be a way for the policy maker to inform the mechanism of changes in policy. The requestors are the source of requests to allocate or free resources. All of the requestors must be at the same layer or higher than the mechanism, but need have no particular relation to the policy maker or each other. All requests are directed to the mechanism, which decides whether to honor them based on the policies of which it is aware. The supplier actually performs the allocation and freeings and inter-

prets all other operations on the resource. In addition to the policies enforced by the mechanism, the supplier may impose restrictions of its own, e.g. it may decide not to allocate a resource unless one is available. The supplier must be at the same layer or lower than the mechanism. It is quite possible to combine the functions of the mechanism and the supplier. The critical factor in the location of the mechanism is that it is between the supplier and all requestors.

Resource control policies can be divided into two classes: one where resources are pre-allocated, and one where resources are allocated only as needed. The first class of policies has the advantages that overhead is small, and, once resources have been allocated, no time need be wasted in waiting for resources to be available. These policies can be shown to be free of deadlock, as long as all resources are requested before any are used. They have the disadvantage that a requestor may not know how many resources it needs, so unnecessary resources may be allocated. The second class of policies is less wasteful of resources, but requires sophisticated methods to avoid deadlocks. The second class includes some very complicated policies because a resource can be implicitly allocated by a requestor by simply using one that it does not have. In the model, one can see instances where both classes of policies are used for different resources. PAGE CONTAINERs are only allocated as needed for PAGEMENTs. However, whenever a PAGE CONTAINER is allocated, a home is assigned to it, regardless of whether the home is ever needed.

Very early in this thesis, problems with the Multics resource control system were encountered (see chapter two). Much of the work here can be described as reorganizing the resource control mechanism so that it will fit neatly into a type extension hierarchy. The reorganization, however, leaves the current guiding policies of Multics intact. The quota system is quite useful. It permits the creation of a set of resource allocation centers (quota cells). Resources may be allocated and charged to a center if the center has enough quota. The problem with the Multics resource control policy is that the centers are tied to the directory hierarchy. The directory hierarchy originated as a naming system. It solved some of the problems of global name spaces and the occurrence of duplicate names. In Multics, the directory hierarchy is much more. In addition to a naming hierarchy, it also embodies the authority hierarchy (access control) and the resource control hierarchy.

Instead, we propose the separation of these hierarchies, as much as possible. Logically, they are independent. Forcing them to be identical creates other problems. For example, a directory which contains a quota cell, against which any active segment is charged, must stay active. We have intentionally designed the bottom two layers of the virtual memory manager so that they make no assumptions about the hierarchies. Therefore, they could be used, without modification, regardless of how the hierarchies are combined. Separate directory and resource control hierarchies would allow segments in many different directories to be charged against the same quota cell, or segments in the same directory to be charged against different quota cells. These facilities are useful so that a user may structure his naming environment as he likes, independent of how his resources are controlled.

7.4 Directions for Future Research and Development

Since the virtual memory manager model is entirely on paper, we have had to appeal to the reader's intuition as to its simplicity and consistency. The acid test of any design is implementation. The existence of a working implementation guarantees that nothing has been ignored or overlooked. While we feel that the model is correct, the only way to prove correctness is to implement it. An implementation would require large amounts of time and computer resources, but would, hopefully, verify our results.

To further simplify the Multics supervisor, we can suggest that the interactions between directory control and segment support be studied. We have asserted that they interact poorly, but we have made no attempt to change them. The interesting problem lies in the relationships among the directory, authority, and resource control hierarchies. These hierarchies serve different functions, so should not necessarily be combined [Rotenberg 1974]. One of the advantages of the model over Multics is that the model makes no assumptions about these hierarchies. Thus, separate directory and resource control hierarchies could be implemented on top of the model, allowing segments in the same directory to be charged to different accounts (quota cells). Another interesting approach would be a multiprocess implementation of segment support and directory control. This could be along the lines followed by Huber [1976].

A process executing on Multics operates in several related address spaces. At times, the process is in an environment where it may only reference primary memory by absolute address. Most of the time, the process is in

a segmented address space, but the segments might be unpaged segments, supervisor segments, or file system segments. The variety of segment types means that the process must operate in at least four different address spaces. How these address spaces overlap and interact is poorly understood. Research in this direction might point out a clearer and more efficient way to manage the sharing of address spaces.

In a more academic vein, another research direction is suggested by chapter three. In that chapter, we spoke loosely of a procedure for modularizing a system. The procedure involved the iterative application of a set of techniques. Projects, such as the one described in this thesis, would be greatly aided by a formal modularizing procedure. However, devising such a procedure is quite difficult. It requires careful definitions of modules and connections, a method for comparing them, and a proof that such a procedure will terminate with a correct result.

We also suggest further study of the need for objects in operating systems. The third difference between Multics and the model, noted in section 7.2.2, arose because we have no semantics for describing an object which is created or allocated by one layer and manipulated by a lower layer. One possible direction for study of this problem is provided by Janson's software cache structure [1976]. The appeal of objects is growing. Since operating systems are fundamental to computer systems, the uses of objects in operating systems should be better understood.

APPENDIX

Summary of Types Used in the Model

Type: PAGE CONTAINER

Attributes:

name

data array

home

used flag

modified flag

zero flag

core flag

Operations:

allocate (home, zero_flag, name)

free (name, zero flag)

read (name, offset, value)

write (name, offset, value)

get_home (name, home)

usedp (name, flag)

modifiedp (name, flag)

zerop (name, flag)

corep (name, flag)

Type: QUOTA CELL

Attributes:

name

frame quota

frames used

time-frame product

Operations:

allocate (quota, used, time-frame_product, name)

free (name, quota, used, time-frame product)

change_used (name, quantity)

reset_time-frame_product (name, product)

move_quota (source_name, target_name, quota_quantity)

move_quota_used (source_name, target_name, quota_quantity,
used_quantity)

Type: PAGEMENT

Attributes:

name

size

length

frames used

QUOTA CELL name

used flag

modified flag

physical volume

core count

page table

page table modified flag

data array

Operations:

allocate (size, physical_volume, page_table, quota cell, name)

free (name, physical volume, page table, quota cell, used flag, modified flag)

read (name, offset, value)

write (name, offset, value)

usedp (name, flag)

modifiedp (name, flag)

page_table_modifiedp (name, flag)

get_page_table (name, physical volume, page table)

get_QUOTA_CELL (name, cell name)

change_QUOTA_CELL (name, new cell name)

get_core_count (name, count)

truncate (name, length)

move_contents (name, size, new name)

REFERENCES

[Bensoussan, Clingen, and Daley, 1972] A. Bensoussan, C. T. Clingen, and R. C. Daley, "The Multics Virtual Memory: Concepts and Design," Communications of the ACM 15, 3 (March 1972), pp. 135 - 143.

[Dahl, Dijkstra, and Hoare, 1972] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Structured Programming, A.P.I.C. Studies in Data Processing No. 8, Academic Press, London and New York, 1972.

[Dijkstra, 1968a] E. W. Dijkstra, "The Structure of the THE-Multiprogramming System," Communications of the ACM 11, 5 (May 1968), pp. 341 - 346.

[Dijkstra, 1968b] _____, "Complexity Controlled by Hierarchical Ordering of Function and Variability," Proceedings of the NATO Science Committee Conference, ed. P. Naur and B. Randell (October 1968), pp. 181 - 185.

[Floyd, 1967] R. W. Floyd, "Assigning Meanings to Programs," Proceedings of Symposium in Applied Mathematics, Volume 19 (ed. J. T. Schwartz) American Mathematical Society, Providence, R. I. (1967), pp. 19 - 32.

[Hoare, 1969] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," Communications of the ACM 12, 10 (October 1969), pp. 576 - 580.

[Huber, 1976] A. R. Huber, "A Multi-Process Design of a Paging System," S.M. and E.E. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, 1976, and M.I.T. Laboratory for Computer Science Technical Report TR-171.

[Hunt, 1976] D. H. Hunt, "A Case Study of Intermodule Dependencies in a Virtual Memory Subsystem," S.M. and E.E. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, and M.I.T. Laboratory for Computer Science Technical Report TR-174.

[Janson, 1976] P. A. Janson, "Using Type Extension to Organize Virtual Memory Mechanisms," Ph.D. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, 1976, and M.I.T. Laboratory for Computer Science Technical Report TR-167.

[Lampson and Sturgis, 1976] B. W. Lampson and H. E. Sturgis, "Reflections on an Operating System Design," Communications of the ACM 19, 5 (May 1976), pp. 251 - 265.

[Liskov, 1972a] B. H. Liskov, "The Design of the Venus Operating System," Communications of the ACM 15, 3 (March 1972), pp. 144 - 149.

[Liskov, 1972b] _____, "A Design Methodology for Reliable Software Systems," Proceedings of the Fall Joint Computer Conference, 1972, pp. 191 - 199.

[Liskov et al., 1977] B. H. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU," to appear in Communications of the ACM 20, 7 (July 1977).

[Naur, 1966] P. Naur, "Proof of Algorithms by General Snapshots," BIT 6, 4 (1966), pp. 310 - 316.

[Organick, 1972] E. I. Organick, The Multics System: An Examination of Its Structure, M.I.T. Press, Cambridge, Massachusetts, 1972.

[Parnas, 1971] D. L. Parnas, "Information Distribution Aspects of Design Methodology," Proceedings of IFIP Congress 71, ed. C. V. Freiman (August 1971), Volume I, pp. 339 - 344.

[Parnas, 1972a] _____, "A Technique for Software Module Specification with Examples," Communications of the ACM 15, 5 (May 1972), pp. 330 - 336.

[Parnas, 1972b] _____, "On the Criteria To Be Used in Decomposing Systems into Modules," Communications of the ACM 15, 12 (December 1972) pp. 1053 - 1058.

[Parnas, 1976] _____, "Some Hypotheses about the Uses Hierarchy for Operating Systems," Research Report BS I 76/1, Technische Hochschule Darmstadt, Fachbereich Informatik (March 1976).

[Popek, 1974] G. J. Popek, "A Principle of Kernel Design," AFIPS National Computer Conference Proceedings, Volume 43, AFIPS Press, Montvale, New Jersey (1974), pp. 977 - 978.

[Reed, 1976] D. P. Reed, "Processor Multiplexing in a Layered Operating System," S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, 1976, and M.I.T. Laboratory for Computer Science Technical Report TR-164.

[Robinson et al., 1975] L. Robinson, K. N. Levitt, P. G. Neumann, and A. R. Saxena, "On Attaining Reliable Software for a Secure Operating System," Proceedings of the International Conference on Reliable Software, and ACM SIGPLAN Notices 10, 6 (June 1975), pp. 267 - 284.

[Rotenberg, 1974] L. J. Rotenberg, "Making Computers Keep Secrets," Ph.D. Thesis, M.I.T., Department of Electrical Engineering, 1974, and M.I.T. Project MAC Technical Report TR-115.

[Saltzer, 1974] J. H. Saltzer, "Protection and the Control of Information Sharing in Multics," Communications of the ACM 17, 7 (July 1974), pp. 188 - 402.

[Schell, 1971] R. R. Schell, "Dynamic Reconfiguration in a Modular Computer System," Ph.D. Thesis, M.I.T., Department of Electrical Engineering, 1971, and M.I.T. Project MAC Technical Report TR-86.

[Schroeder, 1975] M. D. Schroeder, "Engineering a Security Kernel for Multics," Proceedings of the Fifth Symposium on Operating Systems Principles, and ACM Operating Systems Review 9, 5 (November 1975), pp. 25 - 32.

[Simon, 1962] H. A. Simon, "The Architecture of Complexity," Proceedings of the American Philosophical Society 106, 6 (December 1962), pp. 467 - 482.

[Stern, 1976] J. Stern, "Multics Security Kernel Top Level Specification," Draft Project Guardian Report, November, 1976.

[Sturgis, 1974] H. E. Sturgis, "A Postmortem for a Time Sharing System," Ph.D. Thesis, University of California at Berkeley, Department of Computer Science, 1974, and Xerox Palo Alto Research Center, Palo Alto, California, CSL 74-1.

[Wulf et al., 1974] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "HYDRA: The Kernel of a Multiprocessor Operating System," Communications of the ACM 17, 6 (June 1974), pp. 337 - 345.