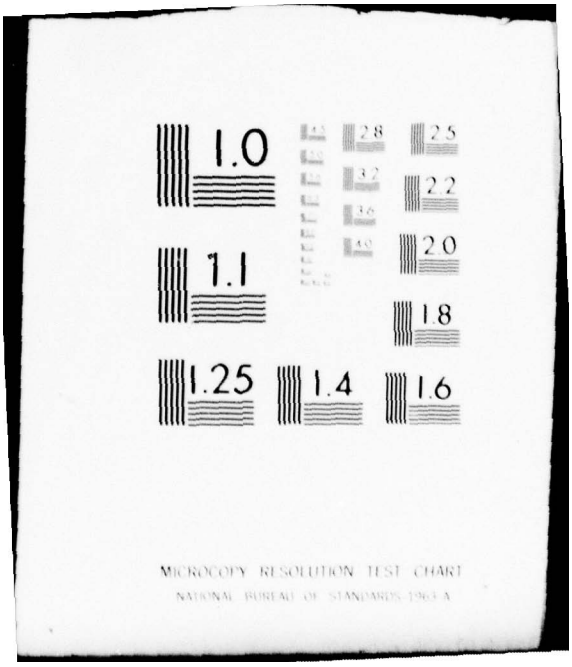
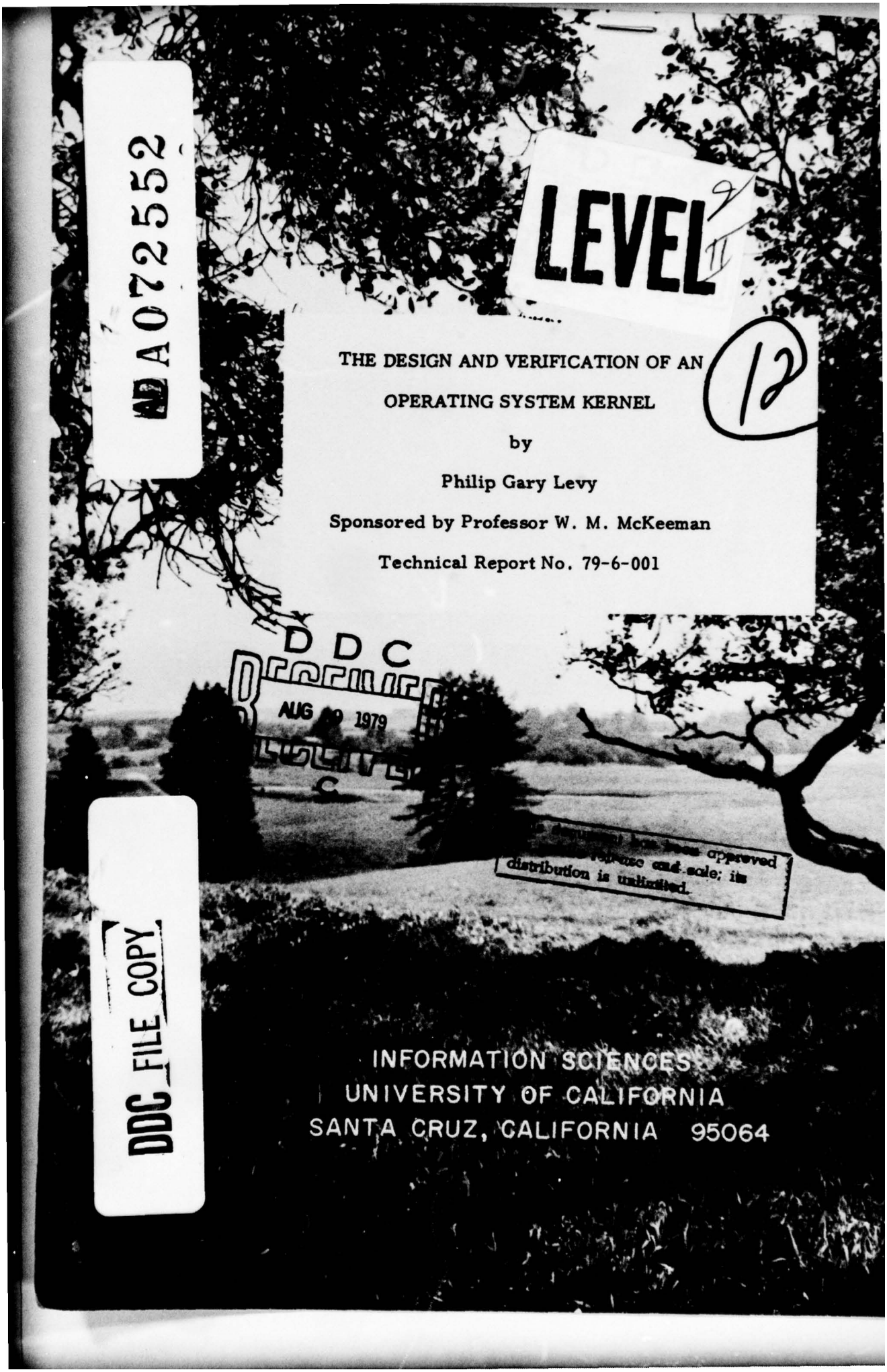
AD-A072 552 CALIFORNIA UNIV SANTA CRUZ INFORMATION SCIENCES F/G 9/2
THE DESIGN AND VERIFICATION OF AN OPERATING SYSTEM KERNEL.(U)
JUN 79 P G LEVY N00014-76-C-0682
UNCLASSIFIED TR-79-6-001 NL

1 OF 2
AD A072552


1.0
1.1
1.25
1.4
1.6
2.8
2.5
3.2
2.2
3.6
2.0
4.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A


AD A072552
LEVEL II
12
THE DESIGN AND VERIFICATION OF AN
OPERATING SYSTEM KERNEL
by
Philip Gary Levy
Sponsored by Professor W. M. McKeeman
Technical Report No. 79-6-001
DDC
AUG 9 1979
C
This document has been approved for public release and sale; its distribution is unlimited.
DDC FILE COPY
INFORMATION SCIENCES
UNIVERSITY OF CALIFORNIA
SANTA CRUZ, CALIFORNIA 95064

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

Masters thesis

4. TITLE (and Subtitle)

The Design and Verification of an Operating System Kernel.

5. TYPE OF REPORT & PERIOD COVERED

Technical / Sept.

6. PERFORMING ORG. REPORT NUMBER

TR-79-6-001

7. AUTHOR(s)

Philip Gary Levy

8. CONTRACT OR GRANT NUMBER(s)

N00014-76-C-0682

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Information Sciences
University of California
Santa Cruz, California 95064

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

Office of Naval Research
Arlington, Virginia 22217

12. REPORT DATE

19 June 1979

13. NUMBER OF PAGES

230 p.

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

Office of Naval Research
University of California
553 Evans Hall
Berkeley, California 94720

15. SECURITY CLASS. (of this report)

Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

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)

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

Practical aspects of the design, implementation, and verification of operating systems are explored in this thesis. Axiomatic techniques for proving parallel programs (developed by Hoare, Owicki, and Gries) are extended and applied in the proof of an operating system kernel. The techniques developed apply directly only to environments with a single physical processor.

A technique for mutual exclusion that is simple to implement and has properties that contribute to verification is presented. It is based on the

(Continued on next page)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

410 350

20. (Continued)

disabling of context switching to allow a single process to execute without pre-emption. In such an excluded region, sequential verification techniques can be applied.

The axiomatic specification technique is extended to deal with predicates that are subject to interference. An extended axiom consists of two parts: one that specifies what the operation actually does (using any predicates), and the other that uses predicates not subject to interference and hence usable in proofs. The use of extended axioms in program proofs is discussed.

The nature of predicates and deductions used in program proofs are discussed. In some contemporary machines, atomic access to variables may not be possible. Several conditions that assure safe access to variables are presented. The roles of mutual exclusion, priority-based scheduling, and point-of-view in establishing freedom from interference are explored.

Finally, the specification and verification of the kernel are presented. The specification technique and verification procedure are described.

The Design and Verification of an Operating System Kernel

Philip Gary Levy

Abstract

Practical aspects of the design, implementation, and verification of operating systems are explored in this thesis. Axiomatic techniques for proving parallel programs (developed by Hoare, Owicki, and Gries) are extended and applied in the proof of an operating system kernel. The techniques developed apply directly only to environments with a single physical processor.

A technique for mutual exclusion that is simple to implement and has properties that contribute to verification is presented. It is based on the disabling of context switching to allow a single process to execute without pre-emption. In such an excluded region, sequential verification techniques can be applied.

The axiomatic specification technique is extended to deal with predicates that are subject to interference. An extended axiom consists of two parts: one that specifies what the operation actually does (using any predicates), and the other that uses predicates not subject to interference and hence usable in proofs. The use of extended axioms in

program proofs is discussed.

The nature of predicates and deductions used in program proofs are discussed. In some contemporary machines, atomic access to variables may not be possible. Several conditions that assure safe access to variables are presented. The roles of mutual exclusion, priority-based scheduling, and point-of-view in establishing freedom from interference are explored.

Finally, the specification and verification of the kernel are presented. The specification technique and verification precedure are described.

UNIVERSITY OF CALIFORNIA

SANTA CRUZ

The Design and Verification of an Operating System Kernel

A Thesis submitted in partial satisfaction
of the requirements for the degree of

MASTER OF SCIENCE

in

INFORMATION SCIENCE

by

Philip Gary Levy

June 1979

The Thesis of Philip Gary Levy
is approved:

C. M. McKeeman

Frank DeRemer

Michael Cunningham

John E. Hall

Dean of the Graduate Division

Acknowledgements

I would like to thank the following individuals:

Bill McKeeman for his hours of advice and criticism, and innumerable other contributions;

Loren Meissner for helping me grow;

David Stonerod for starting me in this field; and

My Parents for making it all possible.

This research was partially supported by Office of Naval Research Contract No. N00014-76-C-0682.

1.1 - Problem Area

This thesis is concerned with practical aspects of the design, implementation, and verification of operating systems. Operating systems have commonly been large and complex programs that were plagued with bugs and were difficult to modify and maintain.

Much work has been done over the last 10 years to improve this situation. Programming methodologies have been proposed to record and structure the design of operating systems. Techniques have been devised and developed that allow the formal proof of the correctness of a piece of program text, correctness meaning consistency with its specification.

We will be concerned primarily with the issue of correctness of operating systems. Of secondary concern are engineering aspects of programs (see literature review). Most of the work in this area is subjective in nature. We will comment on the relation of some of these techniques to the specification and verification task.

With the growing use of computers in sensitive areas such as health care and communications, the issue of the production

of programs that do a "good" job has received much attention. While a precise characterization of "good" is beyond the scope of this thesis, we can give some properties that a "good" program should have. The program should be reliable in that a user should be able to predict what the program will do given a certain request. The program should be characterized by and behave according to some specification so that its behavior can be understood without having to analyze the algorithms themselves. Also, a program should be engineered so that it can be understood and modified without violating its specification, and without excessive effort.

Correctness of a program, then, is only one of many properties that we need. A program should not be designed merely to simplify the verification task; this may complicate the establishment of other properties of the program.

There are always some underlying assumptions about the environment in which a program will run. A common assumption is the reliability and correctness of the machine on which the program executes. While failures of central processors and main memory systems are currently fairly rare, failures of mechanical peripherals are more frequent.

The design of a program can greatly affect its response when some assumption about its enviroment becomes false. Correctness alone will not help the program behave in such a case if, as is typically the case, error situations are omitted from the specification [Parnas 76a].

1.2 - Restrictions

We wish to deal with practical aspects of the verification of a real operating system. By 'real' we mean one that has been implemented and is in use and was not designed with verification in mind. The system was, however, designed with "good" software engineering practices in mind. One would expect a system to be designed in this way since verification is not an end in itself, but only a useful (perhaps very valuable and useful) property of a finished program. Engineering design must deal in a larger scope which includes efficiency or performance considerations, documentation, maintenance, modifications (design changes), and completion schedules.

By 'practical', we mean that we are concerned more with the analysis of predicates and variables used in the proof of the program than with detailed manipulations of the actual

program text. Elimination of clerical errors in the final program is an issue not addressed.

We will consider an environment that has a single central processor. Not all of the results we derive will be applicable to multi-processor systems. Nonetheless, single processor systems represent a significant proportion of operating systems. Network systems may be composed of individual single-processor systems, for example. Even though there are multiple processors in the network each individual node can be dealt with in isolation when we deal with local properties of the system on that node alone.

An operating system kernel for nodes on a network oriented system was implemented in early 1979. This system, hereafter referred to as the Kernel, is implemented on a Z-80A microprocessor system. The system is written primarily in a high level language (PLZ/SYS [Snook et al 78]) although some parts are in assembly language. A subset of the Kernel is specified and verified in this thesis.

1.3 - Work Done in This Thesis

The emphasis of this thesis is the verification of the Kernel. Discussions of most other topics are related to the verification issue.

In section 2, literature in the areas of program verification, parallel program verification, operating system design, and software engineering is reviewed.

Section 3 covers software engineering techniques more thoroughly and relates some of the issues to verification. The notion of binding in a program, a measure of its interconnectivity, is related to the difficulty of verification and maintenance.

The achievement of mutual exclusion by context switch disabling is introduced and developed in section 4. This technique, when applied with the idea of virtual machines, is practical and very simple to implement. It also contributes to the verification process in a positive way.

In section 5, problems of axiomatic specification of non-determinitsic and parallel programs are explored. Variables that can be modified by interrupt programs, for

example, are difficult to use in specifications. The idea of extended axioms is introduced to deal with this problem. An extended axiom consists of two parts: one that specifies what the operation actually does, and one that is usable as a proof rule in the usual sense. The use of extended axioms in program proofs is discussed. This approach will be needed in the verification of the Kernel.

The nature of predicates and deductions used in program proofs is explored in section 6. An assumption that variables can be accessed in an atomic manner is false for many computers. Interference between parallel programs and its effect on axiomatic proofs is explored. In many cases, interference cannot be eliminated and techniques for dealing with this problem are developed. These techniques involve the use of mutual exclusion and priority based scheduling.

The design of the Kernel is presented in section 7. The extended axiomatic approach developed in section 5 is used. Abstract models of the system and characterizations of the operations are developed.

In section 8, the proof methodology and the proof itself are presented. First, consistency of the specifications with some global properties of the system is proved. Then a

mapping from the abstract model to the concrete implementation is presented. Finally, the implementation is shown consistent with the specifications and global properties of the system are verified.

In the conclusion, section 9, observations about the entire thesis are presented along with directions for further research.

2.1 - Verification

Dijkstra discusses the notions of program correctness and proofs of program correctness [Dijkstra 72]. He eloquently argues that "the art of programming is the art of organizing complexity" and that some formal means of verifying the programmer's reasoning is essential. Here he gives his famous quote: "Program testing can be used to show the presence of bugs, but never to show their absence!"

Floyd describes the state of a program using predicates relating its variables and shows how predicates are affected as flowchart boxes are passed [Floyd 67]. Loop termination must be separately argued, typically by showing that some quantity is reduced with each execution of a loop and this quantity cannot be reduced indefinitely.

Hoare expands on this idea and introduces the formalism P{S}Q to indicate that if a predicate P is true before execution of program statement S, then a predicate Q will be true when S terminates [Hoare 69]. The formalism can be used to state axioms that describe the semantics of statements in a programming language. These axioms, when combined with appropriate inference rules and predicates defining the input assumptions form a deductive system in

which predicates characterizing output variables of a program can be proved. Again, termination must be shown separately.

The axiomatic approach has been expanded to include procedures and parameter passing and jumps and functions [Hoare 71a] [Clint & Hoare 72]. In particular, the programming language PASCAL [Jensen 75] has been axiomatized [Hoare 73].

A related approach to semantics and verification is Dijkstra's weakest pre-condition (wp) [Dijkstra 76]. Rather than moving forward across program statements,

wp(S,R)

is the weakest condition required to guarantee that predicate R holds after execution of statement S. Termination of S is required. A verification that predicate T holds after execution of program P involves:

1) Computation of wp(P,T)

and verification that

2) pre(P) => wp(P,T)

where pre(P) is a predicate that is known to hold before execution of P.

Another approach to program verification is denotational

semantics [Scott 71]. Programs are considered sets of recursive functions whose least fixpoints (in the lattice theory sense) define the meaning of the program. A more literal modelling of environments and scopes is also given.

In our verification, we use Hoare's axiomatic approach.

2.2 - Parallel Techniques

Hoare's axiomatic approach is extended by Owicki to include parallel programs [Owicki & Gries 76a, 76b]. Statements S1, ..., Sn are executed in parallel in the construct

cobegin S1 // S2 // ... // Sn coend.

Given the semantics of each Si in isolation, Pi {Si} Qi, the axiom describing the parallel execution of S1 through Sn is

$$\frac{P1\ \{S1\}\ Q1,\ \ldots,\ Pn\ \{Sn\}\ Qn \text{ are interference free}}{P1\ \&\ldots\&\ Pn\ \{\text{cobegin } S1\ //\ \ldots\ //\ Sn\ \text{coend}\}\ Q1\ \&\ldots\&\ Qn}$$

Interference free is defined by (from [Owicki & Gries 76b]):

> Given a proof P{S}Q and a statement T with precondition pre(T), we say that T does not interfere with P{S}Q if the following two conditions hold:
>
> 1) Q & pre(T) {T} Q
>
> 2) For every statement S' within S, pre(S') & pre(T) {T} pre(S').

It may be necessary to add auxiliary variables to a program to construct assertions that are interference free. Owicki shows that such variables are necessary only formally and can be removed from the program without altering its meaning.

Termination of parallel programs is also discussed. In the parallel environment it is necessary to show, in addition to the decrease with each loop execution of some quantity that cannot infinitely decrease, that other proceses increase the quantity only finitely. The issue of deadlock is also discussed.

Flon presents a methodology for the design and verification of operating systems [Flon 77]. Specification techniques are reviewed and the specification of operating systems by use of an abstract representation of the system and axioms for operations is found to be the most appropriate. The axioms are stated in terms of the abstract representation. Invariant predicates describe legal states of the abstract representation. The specifications themselves can be shown consistent by proving that the invariant predicates are, in fact, invariant over execution of each operation.

A mapping from the concrete to the abstract representation

enables verification of the implementation. A process scheduler is specified and verified using this technique. It is assumed to be run in a sequential environment, however.

Dijkstra's weakest pre-condition semantics is extended to the parallel environment by considering all possible interleaved executions of all processes. This combinatorial weakest pre-condition (cwp) approach is claimed impractical because of the large number of paths of execution that must be considered.

Flon also discusses loop termination and deadlock and process synchronization.

We use Owicki's application of axiomatics to parallel programs and Flon's work on specifications.

2.3 - Operating System Design

Mutual exclusion plays an important role in operating system design. Certain computations must be atomic with respect to other computations.

A monitor [Hoare 74, Brinch-Hansen 73] provides exclusion for access to some resource. Monitors for arbitrary resources or groups of resources can be defined. The language concurrent PASCAL includes monitors as a built-in facility [Brinch-Hansen 76].

The idea of process and structuring based on processes is discussed [Horning & Randell 73]. Various interactions of processes such as synchronization and communication are discussed.

Dijkstra decomposes a system into concurrent processes [Dijkstra 68]. Semaphores with operations P and V are presented as a synchronization mechanism.

The process concept and several forms of mutual exclusion are used in this thesis.

2.4 - Software Engineering

Several design techniques have been suggested with the goal of producing "good" systems. Although many of the concepts seen intuitively sound, there has been little success at measuring the effectiveness of the techniques in a

quantative way. An empirical approach has been suggested but its application is difficult due to the many variables and effort required [Weissman 74].

Wirth describes a stepwise refinement methodology in which, through a series of similar steps, an initial general statement of a solution is refined into a final program [Wirth 71]. The various steps in the refinement are transformations of the previous stage and represent some group of design decisions.

Top-down approaches to design and implementation have been discussed. Denning applies the term "top-down" to a tree-like development of all aspects of program development, including design, data structure definition, and code development [Denning 74]. Morgan applies it only to the design process [Morgan 77], and Myers only to the coding and testing process [Myers 73].

The concept of information hiding has received much attention. Although the concept is very general, it has been applied particularly to the area of data structures. The notion of "abstract data types" [Liskov & Zilles 75] has been developed and appears in several recent programming languages [DeRemer & Levy 79]. Parnas observes that module

interconnections are the assumptions that they make about each other [Parnas 71]. He notes that programmers tend to use all the information available to them and therefore information about modules should be restricted. The issue that management should control how information is restricted but technical people should control what information is restricted is also discussed.

In frequently referenced works, Parnas discusses a data-oriented decomposition of a problem where modules are responsible for maintenance of a particular data structure [Parnas 72a, Parnas 72c]. (A KWICSORT program is used as an example.) The specification techniques have been developed further [Parnas 75d].

Dijkstra begins discussion of another type of methodology based on a hierarchy of abstractions [Dijkstra 68]. In his development of the "THE" system, various layers are placed over the hardware, each defining what was later referred to as a "virtual machine." Some abstractions were implemented as processes. A similar design methodology is expanded to include coding, module interconnection, and testing [Liskov 72]. The term "structured programming" is restricted to the discussing of coding, a distinction frequently missed in other works.

Parnas [Parnas 76b] discusses the effects of building a virtual machine hierarchy and how flexibility and capabilities can be lost as higher abstractions are made. He cautions users of this methodology to be careful not to remove a capability that eliminates a useful design or feature. Negative aspects of top-down design procedures and how the virtual machine approach avoids these problems are also discussed.

The handling of unexpected situations plays an important role in the reliability of a program [Parnas 76a, Parnas 75b]. Inclusion of error conditions in specifications has been advocated [Parnas 75d, 76a]. The term "undesired event" was coined to replace "error", avoiding the complaint that "errors should be corrected, not handled" [Parnas 76a].

"Levels" in the hierarchy need not necessarily correspond to abstractions based on the data structures within an operating system [Parnas 76b, Parnas 76c, and Habermann, et al 76]. One "level" might be context switching based on a fixed process structure and a higher level might include process creation and deletion. Thus, the "process management" module spans levels in the virtual machine hierarchy. This allowed interesting and useful subsets of

operating systems to be developed as a group.

Although the engineering aspects of the Kernel are not specifically discussed in this thesis, consideration of the techniques above played an important role in its design and implementation.

This section discusses software engineering aspects of system design and implementation. Also discussed is the issue of specification of operating system programs and the relation of the engineering techniques to verification.

3.1 - The Importance of Structure

Parnas remarks that "structured is a euphemism for 'restricted'" [Parnas 75b]. A structuring technique eliminates possible organizations by providing a framework in which the problem must be fit. Structuring is important because some "frameworks" have valuable properties. These properties facilitate, for example, the decomposition of a programming task allowing many people to work simultaneously and harmoniously on the task. Structuring can also aid the verification task.

In this section, several structuring techniques are described briefly and related to the verification task. Positive engineering aspects of the techniques are also discussed.

3.2 - Virtual Machines

A machine is a facility that provides capabilities to a program. These capabilities include data storage and manipulation. Historically, a machine (in the context of computers) refered to a physical object that could execute instructions and store data. The machine need not be totally implemented in hardware and the term virtual machine was applied to this case.

An operating system provides a virtual machine for use by user programs. The instructions available include some subset of the instructions defined by the host machine and some instructions implemented by the operating system. The operating system might provide more than one kind virtual machine. Different users may need different capabilities.

An operating system can be internally structured using the notion of virtual machines. Starting with the machine defined by the host machine, some new facilities can be implemented that are then available to users of the new "level" and at the same time, some low-level facilities may be removed. The new facilities are new instructions available to the next higher level. At each level, new facilities are provided that are constructed only out of the

facilities defined by the next lower level (lower meaning closer to the hardware).

In this manner, the system is constructed based first on the hardware. The next level is the hardware supplemented with a context switch "instruction". At increasingly higher levels, a simple scheduler might be added, then some basic synchronization mechanisms, then a clock and round-robin scheduler, then some message facility, and so on. Specific examples of decompositions into such levels appear in [Flon 77], [Saxena 76], and [Parnas 76d].

Note that the implementation of these levels is not specified. Dijkstra [Dijkstra 68] structured the THE system so that levels were implemented by processes. This need not be the case, however.

A hierarchy of virtual machines, each one built from the level one lower than it, is thus defined. Each of these levels can be independently specified and verified. This is exactly the situation when the semantics of a higher level language (such as PASCAL) are defined. Programs written in the language are based on the operations that the language provides and verified in terms of the semantics of those operations, independently of the implementation of the

operations provided by the language.

This abstraction greatly simplifies the verification task for sequential programs. For parallel programs, the decomposition is not necessarily as clear [Gries 76]. In the case of a single sequential program, there is no possibility for observation of the internal states that are part of the implementation of a higher level operation. This is not the case for parallel programs where one program may run while another is in the middle of executing some operation. Additional analysis is required to deal with this situation. The subject is further discussed in section 6.

3.3 - Program Families and Subsets

Consideration of the order in which design decisions are made and the direction in which the virtual machine layers are grown leads to the notion of a family of similar programs. As design decisions are made, the system becomes customized toward some particular goal. From a particular point in the design, a system could be grown in many directions and the closure of the set of programs so grown constitutes a family that share common ancestors.

A drawback that obsoletes many operating systems is their lack of flexibility in terms of future change. Parnas gives an eloquent discussion of this issue [Parnas 77]. If thought is given towards developing a family of programs during the entire design process, then future changes can be made more managable. Implications of design decisions can be tracked during the design and implementation of a system. Then, if that decision is changed as the goals of the system change, the affected areas of the design and the implementation can be identified.

3.4 - Data Abstraction

The notion of data abstraction has emerged as one of the most important in the area of programming [Liskov 72, Liskov&Zilles 75, Parnas 72c]. Data abstraction is an implementation technique where implementation decisions about a data type are hidden from the users of that type. Representations can be changed with only local effects; the programming task can be decomposed along abstraction lines facilitating work on a system by several individuals.

The verification task can be broken down along abstraction lines similar to those of virtual machines. The problem of

parallel programs and the observation of internal states in the implementation of an abstraction (which are hidden from the users of the abstraction) is also shared.

3.5 - Binding in Programs

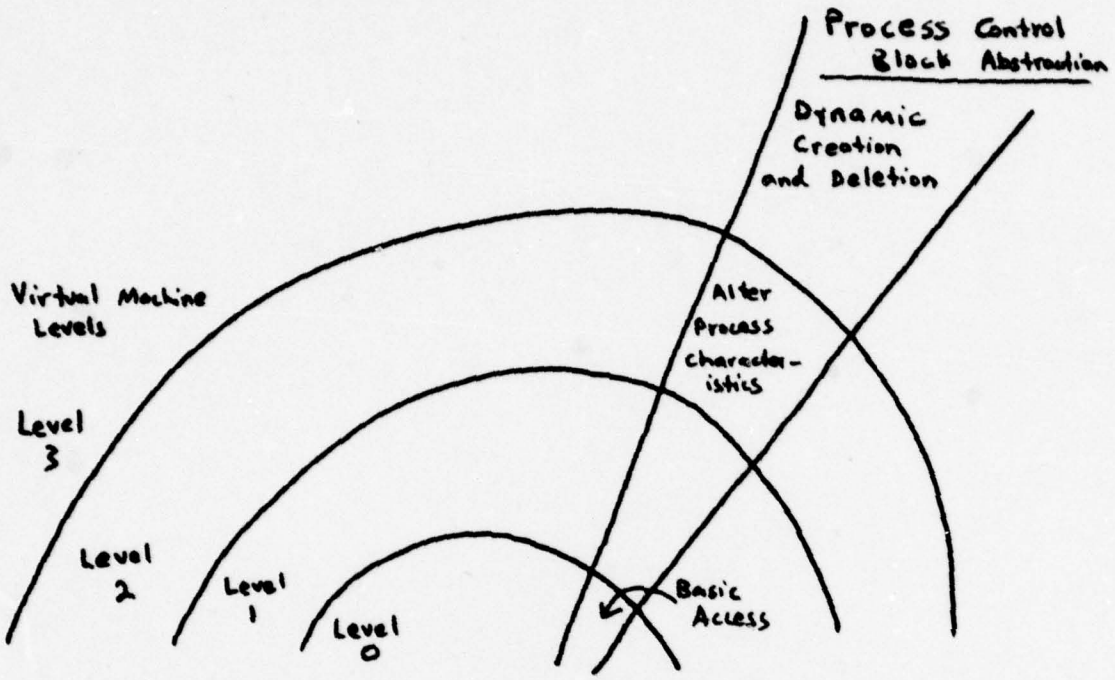
Binding refers to connections between pieces of a program. These connections can be direct: access to variables or calls to procedures where the names of the variables or procedures are given, or indirect: where the access is through a pointer or indirect call via another procedure that has direct access.

Binding where one part of a program refers to a specific name from another part can be dealt with by conventional specification and verification techniques. Implicit binding (for example assumptions that certain variables are modified by one process before being examined by another process) are not specified, cause problems in implementation and maintenance and are difficult to indentify. Implicit bindings are often time related as axiomatic specifications deal with the time dimension poorly.

Binding also is related to the complexity of the

verification task as it is an indication of interconnections in a system. The less that one part of a system can interfere with another, the less that need be considered in the verification. If two subsets of a program are mutually independent, then they can not interfere with each other and can be verified independently.

If we consider the boundaries between layers in the virtual machine hierarchy as horizontal, data abstraction provides similar boundaries except for their vertical orientation. Data abstractions can span virtual machine levels [Habermann et al 76] as in, for example, a Process Control Block abstraction (see figure). Access functions to hardware state stored in the control block would appear low in the VM hierarchy whereas functions for creation and deletion of Control Blocks would likely be at a higher level. Data abstractions tend to reduce the binding in a system and provide a boundary for specification and isolation of the verification of the abstraction from that of the remainder of the system.



3.6 - Specification of Levels and Abstractions

The properties and operations being described in operating system specifications are incomplete in that they are in isolation from a program that uses them. Thus, for example, we cannot show that the operating system is deadlock free; that is a property of the processes which run on it. Destructive interference between user processes cannot necessarily be shown either.

The majority of these problems are in area of termination (and hence deadlock or starvation). If our system has a priority-based scheduling philosophy, we cannot guarantee,

for example, that a process waiting for some signal will execute if that signal is issued. It is possible that higher priority processes will run indefinitely so we can only prove that, in this case, the process waiting for the signal will be eligible to run if the signal does come.

Several specification techniques have been proposed [Liskov&Zilles 76]. Flon analyzes appropriateness of several of these techniques and concludes that the use of predicate transformations (based on axioms) with an abstract representation is most easily applied to operating systems [Flon 77]. Esssentially, an abstract representation of the system data is defined and operations are specified in terms of their effect on this abstract representation.

Paraphrasing Flon, the axioms characterizing the operations take the form:

P(X) { F(X) } Q(X)

with the usual interpretation of such an axiom. The predicates are parameterized indicating their dependence on the abstract representation, X. Given a mapping

A:Concrete Representation -> Abstract Representation,

a verification of the implementation of F would involve a derivation of:

P(A(x)) { implementation of F(A(x)) } Q(A(x))

(where x is the concrete representation such that A(x)=X). The function A must be well defined for the proof to be meaningful. This is insured by establishing invariants describing allowed configurations of the abstract representation. Given such an invariant, I, we must also show that I is in fact invariant across each operation: (for each operation F in the system)

I(X) and P(X) { F(X) } I(X)

This can be done in terms of the specifications alone. Then, any implementation that satisfies the specification will maintain the invariants.

The specifications can be restated in terms of the concrete representation and the verification performed based on these specifications as well.

We must be careful in our specifications for problems that will be caused by the parallelism in our system. As a result, for free variables in the predicates that can be affected by another process, we must use extended axiomatic specifications. This topic is discussed in section 5. Determination of what variables are safely used in predicates is discussed in section 6.

We must also take care in our specifications to avoid

confusion about point of view. An axiom of the form:

Running(Q) & P.priority>Q.priority {WAKE P} Running(P)

may accurately describe a transition in the state of the entire system when a process of higher priority is awoken, but such an axiom is not a valid proof rule in the process Q. In fact, from the point of view of Q, the specificaition should read:

Running(Q) and ... { WAKE P } Running(Q)

It is always the case that, from a particular process' reference frame, it is the running process. In the case of WAKE, "after execution of the operation { WAKE P }" means when execution of Q continues.

Thus, two sets of specifications need to be provided for each layer in the virtual machine: one specifying the semantics from a global view that accounts for all transitions within the system, and one that provides a user of the operations provided on that machine with the semantics from a local view. It is the case that many operations have no local effect, that is they are like "no operation" instructions in their local semantics. The WAKE operation used as an example above is one of these; there is no general way that a process can detect that it has executed the WAKE.

3.7 - Use of the Virtual Machine Model to Control Interference

The virtual machine model can also be used as a tool in limiting the capabilities of a process. Each process runs on some specific virtual machine. Each virtual machine has an instruction set. Processes running on a virtual machine having no instructions that affect a particular variable cannot modify that variable. This technique gives a tool in establishing non-interference among a group of processes.

An example relevant to the operating system we will verify concerns the virtual machine that normal processes run on and another virtual machine that interrupt processing programs run on. The normal processor has no instructions that affect interrupt priorities or control flags; the interrupt processor has only limited capabilities to affect the state of processes running on the normal machine. In particular, interrupt programs can wake up normal processes but may not affect them in any other way.

In this section we describe a low level mutual exclusion technique. The technique is very simple and easy to implement. Its applicablility is restricted to systems with a single physical processor.

Context switching is the reassignment of the processor from one process to another. That is, the hardware context is changed to run another process.

4.1 - Interrupts and Context Switch

Most programmers who have dealt with real-time systems are familiar with the technique of disabling hardware interrupts as an exclusion mechanism. Once interrupts are disabled, only voluntary action by the program can cause a context switch. If the program takes no such action, mutual exclusion is guaranteed. When the interrupt system is again enabled, hardware interrupts can cause context switching. This is the essence of the technique. The problems associated with the missing of real-time events while interrupts are disabled are discussed below.

The general philosophy of this procedure is different from that of most other mutual exclusion techniques. Whereas

most exclusion mechanisms (such as semaphores or monitors) work by acquiring permission to "go ahead" into a critical region, the context switch control works by stopping all other processes from doing anything. The "wait for permission" techniques require some form of queue or busy waiting to keep track of processes that requested but were denied permission to enter a critical region. No such additional mechanisms are needed with context switch control; the scheduler already keeps track of those processes not running.

Freedom from deadlock caused by all processes becomming blocked waiting for entry into an excluded region can be established simply by showing that the code in each critical section terminates and interrupts are enabled on exit. Furthermore, since all other parallel activities are suspended, conventional sequential proof techniques may be applied to a critical section when exclusion is achieved in this way.

We have not solved any of the higher level resource allocation deadlock problems. This technique, however, can be used to implement more complex exclusion mechanisms such as semaphores or monitors.

4.2 - Practicality Problems and a Solution

The question of practicality must still be addressed. The disabling of the hardware interrupt system has an undesirable effect on real-time responsiveness. Use of the virtual machine approach to design described in section 3 helps. We can achieve a certain amount of exclusion by disabling context switching of processes on a particular virtual machine without disturbing the interrupt responsiveness of the system. The extent of the exclusion achieved is reduced, however.

Consider the case of two virtual processors, the I-machine and P-machine. The I-machine runs only interrupt programs and the P-machine runs all other processes. Scheduling in the I-machine is managed predominantly by the hardware but the interrupt system can be disabled or enabled. Scheduling on the P-machine is performed by a scheduling program which runs on the P-machine. When a P-machine program wishes exclusive access to a facility shared only among P-machine processes, it need only disable P-machine context switching to achieve it. I-machine programs still handle hardware

interrupts without interference.

If a facility is shared by programs on both the P- and I-machines, then context switching on both must be disabled to achieve exclusion. In the case a P-machine process wishes to initiate such a critical region, the P-machine context switching is disabled to prevent other P-machine programs from interfering and the I-machine context switching is disabled to prevent the start of any I-machine program. In practice, the size of regions requiring exclusion on both machines is sufficiently small (in time) that interrupt responsiveness is not impaired.

Missing real-time events is the main problem of this technique. Loss of such events, however, is not increased when the virtual processor approach is used. Time-critical events are handled by the I-machine. I-machine programs can queue the events and additional processing of these events can be done on the P-machine so that the exclusion does not interfere directly with the processing of events. In addition, the simplicity of the technique makes more processor time available overall compared to other exclusion techniques.

5.1 - Introduction

This section addresses problems of writing axiomatic specifications for procedures that execute in a parallel environment. Conventional axiomatic specifications have a serious deficiency that makes some supplement necessary.

Hoare uses the notation

$$P \{Q\} R$$

to specify the semantics of an operation Q, namely, that if P is true before the execution of Q, then R is true after the execution of Q [Hoare 69].

The specifications are prototype axioms that are adaptable to specific statements based on the actual content of the statement. An example of this is the assignment statement:

$$P_v^e \{v := e\} P$$

The left predicate is derived from P replacing free occurrences of v with e. A specific case of the use of this rule is:

$$x+1 > 10 \{ x := x+1 \} x > 10$$

An operation can be specified by more than one axiom:

```
                    Count = -1  {Kick}  Count = 0
        Count = 0 | Count = 1   {Kick}  Count = 1
```

These two axioms describe the semantics of an operation "Kick" provided that one of the two preconditions can be established.

Such a specification serves a triple role. First, it provides a specification of the effect of the operation "Kick" without constraining its implementation. The specification also acts as a proof rule usable in a derivation of the correctness of a program that uses the "Kick" operation. Finally, the axioms define the pre- and post-conditions needed to verify the correctness of the implementation of the "Kick" operation.

5.2 - Parallel Programs with Shared Variables

Such specifications are unsound when the context is expanded to a parallel program environment. Suppose predicates from the axiomatic specification of an operation are used in proofs of parallel instruction streams. Variables used in these predicates might be changed by statements in other instruction streams. It is no longer the case that

predicates describing the effect of an operation can be used as assertions about the state of the program that executed the operation.

Consider the following implementation of Kick:

```
Kick PROCEDURE
entry
        if Count<1 then Count := Count + 1 fi
end Kick
```

and the following program. The statement

```
cobegin S1 // S2 // ... // Sn coend
```

indicates parallel execution of each Si [Owicki & Gries 76b]. Now consider a use of Kick:

```
cobegin
        Count := 0
        Kick

//
        repeat case RandomNumber(0..1)
                <.3 :          Count := -1
                >= .3 and <.6:Count := 0
                >= .6 :        Count := 1
        forever
coend
```

It is clear that the two processes interfere in that the shared variable Count is modified by both. Let us suppose that the Kick operation is not interruptable. We are left with the problem that even though the axioms above describe correctly the effect of the procedure Kick, they cannot be used in a proof of this program that uses Kick. This is due to the possibility that immediately after execution of Kick,

the second process changed Count, invalidating the postcondition of the Kick operation. For the same reason, we cannot make any statement about the precondition for the call to Kick.

Any value of Count is possible before or after Kick is executed. The program cannot observe the actual value nor can we formally predict it.

5.3 - Extended Axioms

We are left with the problem that although we have a set of axioms descriptive of the action taken by the procedure Kick, these axioms are not usable as proof rules.

A solution to this problem involves the separation of predicates in the axioms into two groups. One contains axioms characterizing pre and post-conditions that are invariant over the execution of the system, i.e., those that cannot be affected by the execution of other processes. The other group of predicates are those that can be affected by other processes. The general form of an extended axiom is:

```
P { operation } Q

with oneof

        P1 { operation } Q1
        P2 { operation } Q2
              ...
              ...
```

The interpretation of an extended axiom is as follows.

The predicates P and Q must not be affected by any of the operations executable by other concurrent programs. These predicates are useable in axiomatic proofs in the usual sense [Owicki&Gries 76a, 76b].

Exactly one of the Pi predicates must hold at any time. The action characterized by the oneof axiom whose precondition holds will be performed by the operation. Note that it is not generally possible to determine which of the oneof axioms applied in any particular execution of the operation. Nor is it possible to assert the truth of any of the Qi post-conditions. All that can be said of the oneof axioms is that the action specified by one of them was performed.

5.4 - Use of Extended Axioms in Program Proofs

Let's now respecify Kick using extended axioms:

```
-1<=Count<=1 { Kick } -1<=Count<=1

     with oneof

                          Count = -1 {Kick} Count = 0
          Count = 0 | Count = 1  {Kick} Count = 1
```

Again considering

```
cobegin
        Count := 0
        Kick
        if Count = -1 then Print("At least one
                          random number was <.3") fi
//
        repeat case RandomNumber(0..1)
                <.3 :            Count := -1
                >= .3 and <.6:   Count := 0
                >= .6 :          Count := 1
        forever
coend
```

we can now prove the correctness of the output of the first program. After execution of Kick we have:

```
-1<=Count<=1
```

Since Kick was executed, one of the actions specified by the oneof axioms was performed so that at some point

```
Count = 0 | Count = 1.
```

The precondition for the execution of Count := -1 (the only statement that can affect the predicate Count=0 | Count=1) is a random number <.3 and this is the only assignment that can set Count to -1. Therefore execution of the Print

required Count = -1 which required an random number < .3.

Proofs such as this establish global properties of a system of interacting processes. The general form of such a proof involves a case by case examination of the oneof axioms and then consideration of preconditions on statements of other processes that can affect the post-conditions of the oneof axioms. This form of proof is used to establish basic process synchronization properties in the operating system.

An important aspect of the application of the use of extended axioms involves the identification of statements that can affect the post-conditions of the oneof axioms. Techniques for identifying such statements and other issues of interference between processes are discussed further in section 6.

6.1 - Introduction

This section explores the issue of the use of predicates to describe program state in parallel programs. Predicates are assertions about program variables and relations between them.

The issue of access to variables is not addressed by axiomatic proof techniques. Section 6.2 covers this subject. The problem of shared variables has been discussed in section 5 and is further discussed in 6.3. Sections 6.4, 6.5, 6.6, and 6.7 propose techniques for dealing with interference.

6.2 - Assumptions and Restictions Concerning Variable Access

We define a "safe" access to a variable as one that loads or stores an intact value. That is, there is no possibility that the value loaded from a variable is not one that was previously stored.

The following are sufficient conditions for safe access:

1) Access to the variable is atomic. This is a machine dependent condition as, for example, there is atomic access to a 32-bit variable on an IBM 360 but not on a Z-80A. Atomic access might also depend on the form of the access. Statically allocated 16-bit variables in PLZ/SYS are accessed atomically, but local variables in the stack frame are not.

2) Access to the variable appears in a region of the program excluded to all other accessors of that variable.

3) The variable is local to a process. In this case, no other processes can access the variable at all.

4) There are no write accesses to the variable. We assume some initialization prior to the environment where there are multiple accessors.

5) If all processes that read and write the variable can be identified, then some statement about their behavior might be made that would allow a conclusion that an access to a variable is safe.

6.3 - Interference

Some problems with the use of predicates as assertions about the state of parallel programs were discussed in section 5. The general problem concerns predicates changing from true to false as a result of the action of concurrent programs other than the one for which the predicate applies.

This problem is similar in nature to that of aliasing in programming language semantics. There, a variable can change without its name being explicitly mentioned. The deductive system for axiomatic semantics cannot deal with this situation. One approach is to eliminate (forbid) interference. This is analogous to the approach taken in EUCLID [London et al 77]: eliminate aliasing. This won't work for operating systems because some interference is necessary.

Flon notes that for the semantics of languages for parallel programs, possible pre-emption points must be identified [Flon 77]. For example the statement

```
x := x + a
```

must be decomposed into primitive (and atomic) operations

```
x0 := x; x0 := x0 + a; x := x0
```

where x0 is a unique temporary [Flon 77, page 54]. He notes that this rather unfortunately ties down the implementation of such an operation, even on a computer that has an add-to-memory instruction. His combinatorial weakest precondition (cwp) semantics for parallel programs then considers the possible interleaving of executions of the various processes so that modifications of x by other processes will be taken into account.

The combinatorial weakest precondition was found unusable, even in optimized form, for derivations of correctness in the Kernel. The cwp semantics operate on parallel programs that begin and end nicely (in part due to the fact that wp semantics is used to show termination in addition to partial correctness). The Kernel, in contrast, is only a skeleton for user programs. Much of the code is invoked by user programs and we wish to verify the Kernel code independently of them. We need some more powerful way to deal with the problem of other processes changing shared variables.

6.4 - Soundness and Basic Computations

The concept of logical soundness can be applied to axiomatic proofs. A logical system is sound if its axioms cannot be used to derive anything that isn't "true". An axiomatic verification of a program is sound if it does not make a false assertion about the state of a program.

The scope of an assertion is the region of the program where that assertion, or an implication of that assertion, is assumed to hold. It is necessary to show that an assertion is not interfered with in its scope [Owicki & Gries 76a, 76b].

We define a "basic computation" as a sequence of instructions in which there is no voluntary suspension such as a system or supervisor call. It is still possible for a basic computation to be interrupted and execution of concurrent programs to be interleaved with the execution of a basic computation. The context switch that causes this must be based on events not controlled by the instructions of the basic computation.

The points at which a process voluntarily relinquishes

control of the processor are logical points at which to limit the scope of assertions about the state of that process. The process is effectively saying "I'm done with my sub-task. It's all right to let someone else go". We will examine conditions that guarantee soundness of assertions within basic computations.

Soundness of the deductive system in a sequential environment is required for soundness in a parallel environment. Sequential soundness is assumed in the following discussion.

Freedom from interference from other processes is a sufficient condition for soundness. Since no other processes affect any of the predicates within the proof of a basic computation, actions performed by that basic computation can be proved using standard sequential techniques.

Other conditions that imply soundness in a basic computation are explored in the next three sections.

6.5 - Mutual Exclusion and Its Effect on Soundness

Various forms of mutual exclusion can guarantee soundness within a basic computation. If all context switching is disabled then no interference is possible and sequential proof techniques are sound. The exclusion required must exclude processes that can affect variables in the predicates that appear in the proof of the basic computation being considered.

There is an interesting relation between the definition of basic computation and excluded region. When exclusion is released (eg, a V operation), the process is voluntarily allowing itself to be preempted by other processes. In this case, the release of exclusion is the instruction that delimits the basic computation. Thus, the excluded region that manages a resource is a basic computation.

6.6 - Priority Based Scheduling and Its Effect on Soundness

A scheduling philosophy in which processes are assigned distinct priorities and higher priority processes are always run before lower priority processes provides a useful tool

for establishing soundness.

Suppose we are given one process that has write access to a variable and several higher priority processes that have read (but not write) access. Then proofs of basic computations of the higher priority processes are sound with respect to the variable in question. This follows from the scheduling property that runs the reader processes until they voluntarily suspend before running the writer process. Any inspection of the variable and action based on its value that are part of the same basic computation will be completed before the lower priority process is given a chance to alter the variable.

A voluntary suspension by a reader process gives the lower-priority writer a chance to alter the variable, thus delimiting the basic computation.

Access to the shared variable by the writer and readers must be shown safe as part of a verification.

6.7 - Local Freedom from Interference

A weaker condition than Owicki's freedom from interference is sufficient to insure soundness.

Consider the proof of a particular process, process A. Statements in another process may interfere with this proof in that they change variables that falisify assertions in the proof. However, if these changes are reversed before process A is again run, the changes are not observable in process A and do not interfere with its correct funtion.

As long as changes to variables are not observable by a process, then, there is no interference. We call assertions that are subject to interference that is always removed before it becomes observable 'locally interference-free'.

7.1 - Introduction

In this section we present a specification of the Kernel. First, the Kernel is discussed informally. The specification technique is then described. Finally, the specifications are presented and discussed.

The Kernel is designed to provide a multiprogramming environment for microprocessor systems. The Kernel fulfills two primary functions: providing for the sharing of the CPU by "friendly" programs, and providing a standard means for communication between programs.

The hardware consists of a Z-80A microprocessor [Zilog 77], main memory, and various peripheral devices. The Z-80A architecture provides no protection of any kind; all instructions and the entire 64K byte address space are available to any program. A vectored interrupt facility handles the automatic execution of I/O interrupt handlers on a fixed priority basis.

The Kernel is not responsible for I/O and interferes with it to a minimal extent. User programs must handle all I/O devices even if interrupt driven.

The Kernel manages the CPU resource only. Higher levels in the system might provide memory and I/O management. Process are run based on the scheduling philosophy: "run the highest priority, ready process". Facilities are provided so that processes can block or awaken each other. A process is not eligible to run if it is "blocked".

Further facilities in the Kernel provide for the creation and deletion of processes and for the alteration of process priority. A clock facility and round-robin scheduler share time among processes of equal priority. The Kernel's hierarchical structure places these operations at different levels and the Kernel is structured so that all these facilities need not be present.

A program running on the Z-80A has at its disposal 14 8-bit registers, 2 16-bit index registers, a program counter, and a stack pointer. Pairs of 8-bit registers can be used for limited 16-bit operations. A stack facility is provided that allows pushing and popping of 16-bit numbers (from register pairs) and automatic stacking of procedure call return addresses. The interrupt system requires that the stack pointer always point to a usable stack when interrupts are enabled as the hardware interrupt sequence involves pushing context information on the stack.

Due to the intrinsic lack of protection, the Kernel makes no attempt to restrict access to any of its system calls. User programs are required to behave in a friendly way and not destructively modify programs in memory. This environement limits the reliability of the system in that damage done by malfunctioning programs cannot be contained.

The verification of a system composed of user programs and the Kernel involves verification of each component program and verification that no destructive interference exists. It also involves the establishment of non-overflow of runtime allocations (such as the procedure call stack or dynamic variables). This non-overflow condition is an important aspect of program reliability and is also rarely formalized in the language semantics in which programs are verified.

7.2 - The Form of Specifications

The specification of the Kernel consists of the following parts:

1) An Abstract Representation [Flon 77]. This

representation provides a model of the Kernel. There is a mapping from this abstract representation to the concrete representation (used in the implementation). Any state that the Kernel can reach can be characterized by the state of the abstract representation. The specification of the operations that the Kernel provides are stated in terms of their effect on the state.

2) A Set of Virtual Machines. Conceptually, each of these machines runs concurrently although they are all implemented on a single physical processor. Each machine has an instruction set that is not necessarily disjoint from other machines. There may be some rules about relative priority of the various machines.

3) Axioms Describing Each Operation from a Global Point of View. For each operation on each machine, we state axiom(s) that describe its effect on the abstract representation of the Kernel. These axioms are extended as discussed in section 5. They characterize transitions in the Kernel between processes and are used to supply verification conditions for the proof of the implementation of each operation.

4) Axioms Describing Each Operation from a Local Point of View. For each operation, we also describe its effect from the point of view of a process executing the instruction. ("Process" is defined below.) It is these semantics that are needed to verify a user program.

5) Invariants. There are predicates that characterize the legal configurations of the abstract (and concrete) representations. Some constraints on the range of certain variables in the abstract representation are implicitly given by declarations of type or range in the specification of the abstract representation.

6) Theorems. These theorems describe global properties of the system. The theorems generally deal with aspects of interactions among processes that cannot be part of the axioms describing the operations themselves.

The specification of the operations use the extended axiomatic technique described in section 5. Normally, axioms are of the form:

P {S} Q

which reads, "if P is true before execution of S, then Q will be true after execution of S has completed." To simplify the specification, we will list only the variables and predicates that change by the execution of S. For example,

{S} is equivalent to P {S} P

for all predicates P. That is, S has no observable effect.

x=v1 {S} x=v2

indicates that only the variable x changes by execution of S. Thus, any predicate not including a free occurence of x would be invariant across the execution of S.

7.3 - Kernel Specifications

7.3.1 - Overview

Multiprogramming in the Kernel is based on the process model [Horning & Randell 73]. A process is informally an entity that can execute instructions, has a name, and has certain state variables. The precise definition is given in the next section.

Processes execute hardware defined instructions or Kernel defined instructions. We may consider the hardware

augumented by the Kernel as a virtual machine with an enhanced instruction set. The only way that variables in the Kernel's data structures can be accessed is by the Kernel defined instructions.

We can restrict some programs by allowing them to use only a subset of the instructions defined by the Kernel. The Kernel can, thus, provide more than one virtual machine, each with different characteristics, on which user programs may run. Specifically, two machines are so provided: the P-machine and the I-machine.

The P-machine provides instructions for process synchronization (BLOCK and WAKE), instructions for P-machine context switch disabling (FREEZE and THAW), and instructions for controlling the interrupt system (EI and DI). Of these, the I-machine shares only EI and DI. The I-machine also provides an instruction to wake up P-machine processes (IWAKE), and instructions to terminate interrupt handler programs (RETICS and RETI). The instructions EI, DI, and RETI are implemented in hardware on the Z-80A. The P and I-machines are both implemented on a single Z-80A processor.

I-machine programs have absolute priority over P-machine programs when the interrupt system is enabled. On each

machine, a priority based scheduling philosophy determins the process to run. The scheduling is implemented in hardware on the I-machine and by the Kernel on the P-machine. Context switching between processes can be disabled on the P-machine by using the FREEZE instruction. Context switching on the I-machine can be disabled by using the DI (disable interrupts) instruction. THAW and EI enable the respective switching capabilities. If the P-machine executes a DI, context switching between the P and I-machines is disabled so that the P-machine remains in control.

The Kernel maintains state variables that record the condition of each P-machine process. Both P and I-machine instructions can affect these state variables. The instructions are BLOCK, WAKE, and IWAKE.

7.3.2 - Abstract Representation

The Kernel abstract representation consists of the following:

S	The set of P-machine processes
IS	The set of I-machine interrupt handler programs
AM	The active virtual machine (oneof 'P' or 'I')
RPID	The running process on the P-machine (nil if none)

IPID The running program on the I-machine (nil if none)

ION Interrupt system enabled (boolean)
Freeze P-machine context disable call count (non-negative integer)

RSCH Run-the-scheduler flag (boolean)
REDO Scheduler-decision-questionable flag (boolean)

The model of a P-machine process is:

Description	Name	Type
a name, P		
a priority	P.pri	0..255 (255 is highest)
a status	P.st	oneof (B, R, RW)
a state	P.state	machine context

The model of an I-machine program is:

a name, I		
a priority	I.pri	(some ordered set)
a status	I.st	oneof (idle, active)
an event	I.ev	boolean
a state	I.state	machine context

7.3.3 - Virtual Machines

There are two virtual machines. The P-machine runs all processes and the I-machine runs all interrupt programs. The instruction sets of the P- and I-machines are:

P-Machine:

{CS, DI, EI, FREEZE, THAW, BLOCK, WAKE} U IS - {RETI, RETN}

I-Machine:

{IWAKE, RETICS, RESET, DI, EI, RETI} U IS

where

IS = all Z-80A instructions

One component still needs to be added: the process by which devices request interrupt service on the I-machine. We can consider another virtual processor to represent all devices for this function. It has an instruction INTR I, or "interrupt request" for each device I. The specification for these instructions are given in figure 7.5.

7.3.4 - Operation Axioms: Global

The axioms that describe the effect of the execution of the instructions on the P-machine and I-machine are listed in figures 7.1 and 7.2. They are discussed in section 7.4.

7.3.5 - Operation Axioms: Local

The axioms that describe the effect of each P and I-machine instruction from the point of view of a process that executed one of them are listed in figure 7.3 and 7.4.

Note that it is always true that the running process is the same. Some operations have no observable effect. These actually involve global properties of interactions between

processes and their effects are characterized by theorems.

It is recommended that the first-time reader skip to section 7.4 and read it while refering to figures 7.1 - 7.5.

```
The variables P, Q, T range over S (P-machine processes)
         and I, J, K range over IS (I-machine programs)

P1       Freeze=0          {CS}

         with oneof
P1a                        {CS}     PSEL(RPID) & ION & ~RSCH & ~REDO


P2                         {DI}     ~ION

P3                         {EI}

         with oneof
P3a      ION                         {EI}
P3b      ~ION & ∀ J: ~J.ev           {EI}  ION
P3c      ~ION &                      {EI}  J.st=active & IPID=J & AM='I'
         J.ev &
         (∀ K:K.pri>J.pri => ~K.ev)

P4       Freeze=k & k<100 {FREEZE}  Freeze=k+1

P5       Freeze=0          {THAW}

P6       Freeze=1          {THAW}   Freeze=0

         with oneof
P6a                        {THAW}   RPID=T & PSEL(T) & ION &
                                    ~RSCH & ~REDO

P7       Freeze=k & k>1    {THAW}   Freeze=k-1

P8       Q<>RPID           {BLOCK Q}

         with oneof
P8a      Q.st=B | Q.st=R   {BLOCK Q}   Q.st=B & ION
P8b      Q.st=RW           {BLOCK Q}   Q.st=R & ION

P9       Q=RPID            {BLOCK Q}   Freeze=0

         with oneof
P9a      Q.st=B | Q.st=R   {BLOCK Q}   Q.st=B & RPID=T & PSEL(T) &
                                       ~RSCH & ~REDO & ION
P9b      Q.st=RW           {BLOCK Q}   Q.st=R & RPID=T & PSEL(T) &
                                       ~RSCH & ~REDO & ION

P10      Freeze=0          {WAKE Q}

         with oneof
P10a     Q.st=R | Q.st=RW  {WAKE Q}   Q.st=RW & RPID=T & PSEL(T) &
```

			~RSCH & ~REDO & ION
P10b	Q.st=B	{WAKE Q}	Q.st=R & RPID=T & PSEL(T) & ~RSCH & ~REDO & ION
P11	Freeze<>0	{WAKE Q}	
	with oneof		
P11a	Q.st=R | Q.st=RW	{WAKE Q}	Q.st=RW & ION & (~RSCH => (RPID=T & PSEL(T)))
P11b	Q.st=B	{WAKE Q}	Q.st=R & ION & (~RSCH => (RPID=T & PSEL(T)))

The predicate PSEL(P) is

P.st<>B & ∀ Q: Q.pri>P.pri => Q.st=B

Figure 7-1. The P-machine global specifications

```
The variables P, Q, T range over S (P-machine processes)
          and I, J, K range over IS (I-machine programs)

I1      Freeze=0 &          {IWAKE Q}
        RPID<>nil &
        Q.pri>RPID.pri

        with oneof

I1a     Q.st=R | Q.st=RW {IWAKE Q}  Q.st=RW & J0.ev
I1b     Q.st=B           {IWAKE Q}  Q.st=R  & J0.ev

I2      Freeze=0 &          {IWAKE Q}
        RPID<>nil &
        Q.pri<=RPID.pri

        with oneof

I2a     Q.st=R | Q.st=RW {IWAKE Q}  Q.st=RW
I2b     Q.st=B           {IWAKE Q}  Q.st=R

I3      Freeze<>0 &         {IWAKE Q}   RSCH
        RPID<>nil

        with oneof

I3a     Q.st=R | Q.st=RW {IWAKE Q}  Q.st=RW
I3b     Q.st=B           {IWAKE Q}  Q.st=R

I4      RPID=nil            {IWAKE Q}   REDO

        with oneof

I4a     Q.st=R | Q.st=RW {IWAKE Q}  Q.st=RW
I4b     Q.st=B           {IWAKE Q}  Q.st=R


I5      IPID=J0             {RETICS}

        with oneof
I5a     Freeze=0 &          {RETICS}    IPID=nil & J0.st=idle &
        (∀ K<>J0:                       AM='P' & RPID=T & PSEL(T)
         K.st=idle)                     & ~RSCH & ~REDO & ION

I6      I=IPID              {RESET}

        with oneof
I6a                         {RESET} ~I.ev

I7                          {DI}    ~ION
```

```
I8                          {EI}

        with oneof
I8a     ION                   {EI}
I8b     ~ION & ISEL(IPID)     {EI}   ION
I8c     ~ION & J.ev &         {EI}   IPID=J & J.st=active & ISEL(J)
        J.pri>IPID.pri &
        (∀ J:K.pri>J.pri =>
          ~K.ev )

I9      IPID<>J0              {RETI}
        with oneof
I9a     ION & I=IPID          {RETI}  AM='P' & I.st=idle & IPID=nil
        (∀ J<>I:
           J.st=idle)

I9b     ION & I=IPID &        {RETI}  I.st=idle & IPID=J & ISEL(J)
        J.st=active &
        J<>I & (∀ K<>J:
         K.pri>J.pri => (K.st=idle | K=I))
```

The I-machine scheduling predicate, ISEL(I), is:

```
    I.st=active & ∀ J: J.pri>I.pri => J.st<>active
```

Figure 7.2. I-machine global specifications.

L1		{CS}	ION
L2		{DI}	~ION
L3		{EI}	ION
L4	Freeze=k & k<100	{FREEZE}	Freeze=k+1
L5	Freeze=0	{THAW}	
L6	Freeze=1	{THAW}	Freeze=0 & ION
L7	Freeze=k & k>1	{THAW}	Freeze=k-1
L8	Q<>RPID	{BLOCK Q}	ION
L9	Q=RPID	{BLOCK Q}	ION & Freeze=0
L10		{WAKE Q}	ION

Figure 7.3. Local semantics for P-machine operations.

IL1		{IWAKE Q}	
IL2	IPID=J0 & ION	{RETICS}	false
IL3		{RESET}	
IL4		{EI}	ION
IL5		{DI}	~ION
IL6	ION	{RETI}	false

Figure 7.4. Local semantics for I-machine operations.

```
                    {INTR J}

with oneof

ION & AM='P'        {INTR J}  J.ev & ~ION & J.st=active &
                              IPID=J & ISEL(J) & AM='I'

ION & AM='I' &      {INTR J}  J.ev & ~ION & J.st=active &
J.pri>IPID.PRI                IPID=J & ISEL(J)

ION & AM='I' &      {INTR J}  J.ev
J.pri<=IPID.pri

~ION                {INTR J}  J.ev
```

Figure 7.5. Device interrupt request specification.
The variable J ranges over IS (I-machine programs)

7.3.6 - Invariants

The invariants characterize legal states of the abstract representation. We can embody the scheduling philosophies of the P- and I-machines in this way by making invariant the predicate that "the highest priority ready process" is the one that is running (except under certain conditions).

Invariants for the P-machine always hold from the point of view of programs that execute on the P-machine (though not necessarily the implementations of the P-machine operations). The same is true for I-machine invariants and programs.

P-machine Invariants

PIV1 (RPID<>nil & Freeze=0) => (~RSCH & PSEL(RPID))

PIV2 (RPID<>nil & ~RSCH) => PSEL(RPID)

I-machine Invariant

IIV1 ION => (ISEL(IPID) | IPID=nil)

7.3.7 - Theorems

These theorems describe properties that cannot be characterized in the axioms. They involve properties of interactions between processes.

Theorem 1. If a BLOCK P is executed, P will not run until a WAKE P or IWAKE P is executed.

Theorem 2. If no WAKE P or IWAKE P operations are executed, a process P will not run after at most 2 BLOCK P operations are executed.

Theorem 3. A P-machine context switch will not be performed unless Freeze=0 (ie, no context switch disable exists).

7.4 - Discussion of Specifications

P-machine Instructions

In this section, each of the specifications will be briefly discussed. The reader is refered to figures 7.1 - 7.4 for the statement of the axioms describing the operations.

The context switch instruction (CS) is described in axiom P1. Before it can be executed, context switching on the P-machine must be enabled (Freeze=0). It changes the running process on the P-machine to the highest priority ready process, as defined by the scheduling philosophy

predicate, PSEL. The rerun scheduler flag (RSCH) and reconsider scheduling decision flag (REDO) are set to false, but the I-machine can affect these variables so that their values cannot be asserted as part of the main axiom.

P2 and P3, P3a, P3b, and P3c characterize the hardware implemented enable and disable interrupt instructions (DI and EI). The interrupt-on flag (ION) can be affected nondeterministicly by external events so that the value of ION cannot be asserted directly in P3. When interrupts are disabled, this cannot happen so that an assertion of ~ION is interference-free.

FREEZE and THAW, the P-machine context switch disable and enable instructions are characterized in axioms P4 to P7. FREEZEs can be nested and the variable Freeze records the nesting level. Note that the maximum nesting level is 100; the action taken by FREEZE with Freeze greater than this is left undefined. The actions of THAW when Freeze=0 or Freeze>1 are straightforward. When a THAW causes a transition back to a state where context switching is enabled, we assert that the highest priority ready process will begin execution. Again, the values of RPID, RSCH, and REDO are not stable so that values of these variables cannot be predicted.

BLOCK is broken into two cases. In one, a process other than the running process is blocked. There is no transition in the system in this case; only that process' state is changed. When the running process is blocked, any FREEZE conditions are removed and the next process to run is chosen according the scheduling philosophy. In each case, the actual values of the state variables of the process that is blocked are not predictable because the I-machine may affect them at any time. Some transition in the process' state is made and these transitions are characterized by axioms P8a, P8b, P9a, and P9b.

The effect of WAKE on a process' state is treated similarly in axioms P10, P10b, P11a, and P11b. During a FREEZE, no change is made in the choice of running process, but the variable RSCH records the possibility that the running process might not be the one consistent with the scheduling philosophy.

I-Machine Instructions

The IWAKE instruction has similar effect to the WAKE instruction on process state, but in the case of a desired change in the choice of running process, IWAKE sets J0.ev so

that a special interrupt will occur. J0 is an interrupt program that has the lowest priority of all interrupt programs; it runs only when there is no other interrupt processing to be done. As it is an I-machine program, it also runs before the P-machine regains control of the physical processor. J0 executes the RETICS instruction that then forces the proper transition on the P-machine so that the process that runs will be the highest priority ready process.

If IWAKE is executed while Freeze<>0 (axiom I3), it sets RSCH to indicate that the scheduler should be run when a THAW is executed with Freeze=1. It is also possible for the IWAKE to be issued while the scheduler is running (I4). In this case, IWAKE sets REDO to indicate that the scheduler may have made the wrong decision.

The RESET instruction (I6) clears the event flag for the interrupt program that executes it. Again, the flag could be set at any time so that ~I.ev cannot be asserted in general.

I7 and I8 describe the enable and disable interrupt instructions that execute on the I-machine. These are implemented in hardware and are similar in semantics to

their P-machine counterparts.

RETI (return from interrupt) is executed when an interrupt program has concluded the processing of its interrupt. If there are no other interrupt programs in the active state, control returns to the P-machine (I9a). Otherwise, control goes to the highest priority interrupt program that is active (I9b).

Invariants

The P-machine invariant PIV1 states that when there is no rescheduling freeze, the scheduling philosophy applies and the rerun scheduler flag is false. The factor RPID<>nil extends the applicability of the predicate so that when the I-machine is active, the predicate will hold even if RPID=nil.

PIV2 characterizes the relation of RSCH to the scheduling philosophy. As long as RSCH is false, the scheduling philosophy is in effect. When RSCH is true, the scheduling philosophy may or may not hold.

The I-machine invariant IIV1 characterizes the I-machine scheduling philosophy. When interrupts are enabled, either

the highest priority active interrupt process is running, or no interrupt process is running. The invariance of this relation is guaranteed by the hardware.

8.1 - Introduction and Overview

Several steps are involved in verifying the Kernel. Some steps examine properties of the specifications alone, while others deal with the implementation of the Kernel. The predicates used in the proofs also need to be examined for safety and non-interference.

The Kernel is written mostly in PLZ/SYS [Snook et al, 78] and partly in assembly language for the Z-80A. The algorithms are all expressed in PLZ/SYS for the purpose of the verification.

Only the central algorithms of the Kernel are examined. Code that checks for errors in user parameters, for example, is not verified. Nor are we seeking to eliminate clerical errors from the program.

The proof will proceed as follows. First, the invariant predicates will be shown to be invariant over each of the operations (which are assumed to be atomic). For each operation, op, and each invariant predicate I, we must show that given the specification for op:

P { op } Q

we have

$$I \ \& \ P \Rightarrow \exists\ k: P \ \& \ \forall\ x\ (Q \Rightarrow I)$$

where x is the list of variables free in P, Q that can be different in Q than in P and k is the list of variables free in P, Q but not in x [Flon 77]. Then, any implementation that satisfies the specification will maintain the invariants [Flon 77]. The verification of the implementation must establish that the invariants hold at any possible pre-emption points within op.

For example, in

Freeze=1 {THAW} Freeze=0

with oneof

{THAW} RPID=T & PSEL(T) & ~RSCH & ~REDO

x = Freeze, RPID, RSCH, REDO

k = (empty)

The simplified form of the invariant rule when k is empty is

$$I \ \& \ P \Rightarrow \forall\ x\ (Q \Rightarrow I),$$ or specifically,

$$I \ \& \ Freeze=0 \Rightarrow \forall Freeze, \forall RPID, \forall RSCH, \forall REDO$$
$$((Freeze=0 \ \& \ RPID=T \ \& \ PSEL(T) \ \& \ \sim RSCH \ \& \ \sim REDO) \Rightarrow I)$$

Second, the local semantics must be shown to be consistent with the global specifications. If local and global specifications are given:

Pl { op } Ql (local)

Pg { op } Qg (global)

then we must show that no path of execution can invalidate

the implication:

Qg => Ql

This is done by showing that Qg is invariant across all other operations or by giving some other argument that the implication holds.

Next, a mapping from the abstract representation to the concrete implementation is established. Access characteristics of the implementation variables are examined. Then, the code itself is shown to be consistent with the global specifications. This establishes the invariant predicates across each of the operations and the validity of the local specifications. In addition to conventional axiomatic proofs, arguments are presented that access to variables in the predicates in the proofs is safe and consistent.

Finally, the theorems are proved. This step once again involves only the specifications. The theorems will hold for any implementation that is consistent with the specifications.

8.2 - Consistency of Specifications

We wish to establish three properties of the specifications: first, that the predicates PIV1, PIV2, and IIV1 are invariant across each of the operations on the P and I-machines. This aspect of the verification was discussed in section 3.6.

Second, we must show that the post-conditions used in the specifications are interference-free. That is, the predicates of the post-conditions (other than the oneof axioms) must be invariant over operations that can be executed without explicit invocation by the process that executed the original operation. These are the operations on the I-machine (or the I- and P-machines if the I-machine can cause a context switch on the P-machine).

Third, we will establish that the local semantics are consistent with the global semantics. It must be the case that the post-condition in the local specification of an operation is implied by the post-condition of the global specification, and that this implication is not interfered with by any of the other operations. If the operation results in a change of the running process, then the post-condition in the local specification must be implied by

the post-conditions of whatever operations can eventually cause that process to resume execution.

PIV1 (RPID<>nil & Freeze=0) => (~RSCH & PSEL(RPID))

PIV2 (RPID<>nil & ~RSCH) => PSEL(RPID)

IIV1 ION => (ISEL(IPID) | IPID=nil)

PIV1 is invariant on the P-machine.

Axiom	Post-condition	Implies PIV1
P1	PSEL(RPID) & ~RSCH & Freeze=0	Immediate (1) (2)
P2	-	No effect on PIV1 (3)
P3	-	No effect on PIV1
P4	Freeze=k+1 & k+1<>0	Immediate
P5	-	No effect on PIV1
P6	Freeze=0 & RPID=T & PSEL(T) & ~RSCH	Immediate
P7	Freeze=k-1 & k>1	Immediate
P8	-	No effect on PIV1 (4)
P9	Freeze=0 & RPID=T & PSEL(T) & ~RSCH	Immediate
P10	RPID=T & PSEL(T) & Freeze=0 & ~RSCH	Immediate
P11	Freeze<>0	Immediate
I1	J0.ev	Forces execution of RETICS (I5) (5)
I2	Freeze=0 & RPID<>nil	(6)
I3	Freeze<>0 & RSCH	Immediate
I4	RPID=nil	Immediate
I5	RPID=T & PSEL(T) & ~RSCH & Freeze=0	Immediate
I6	-	No effect on PIV1
I7	-	No effect on PIV1
I8	-	No effect on PIV1
I9	-	No effect on PIV1

Notes:

(1) Immediate means that the implication follows by only simplification.

(2) RPID<>nil always holds for programs running on the

P-machine.

(3) No effect means that no variables in the predicate are changed; the implication follows immediately.

(4) When Q<>RPID, PSEL(RPID) is unaffected by a BLOCK Q. The other variables in PIV1 are unaffected also.

(5) Before control returns to the P-machine, the RETICS instruction must be executed. Its effect establishes the invariant.

(6) When Q.pri<RPID.pri, IWAKE Q does not affect PSEL(RPID). The other variables in the invariant are unaffected also.

PIV2 is invariant on the P-machine

Axiom	Post-condition	Implies PIV2
P1	PSEL(RPID) & ~RSCH	Immediate
P2	-	No effect on PIV2
P3	-	No effect on PIV2
P4	-	No effect on PIV2
P5	-	No effect on PIV2
P6	RPID=T & PSEL(T) & ~RSCH	Immediate
P7	-	No effect on PIV2
P8		No effect on PIV2 (4)
P9	RPID=T & PSEL(T) & ~RSCH	Immediate
P10	RPID=T & PSEL(T) & ~RSCH	Immediate
P11	~RSCH => PSEL(RPID)	Immediate
I1	J0.ev	Forces execution of RETICS (I5) (5)
I2	RPID<>nil	(6)
I3	RSCH	Immediate
I4	RPID=nil	Immediate
I5	RPID=T & PSEL(T) & ~RSCH	Immediate
I6	-	No effect on PIV2
I7	-	No effect on PIV2
I8	-	No effect on PIV2
I9	-	No effect on PIV2

IIV1 is invariant on the I-machine

Axiom	Post-condition	Implies IIV1
I1	-	No effect on IIV1
I2	-	No effect on IIV1
I3	-	No effect on IIV1
I4	-	No effect on IIV1
I5	ION & IPID=nil	Immediate
I6	-	No effect on IIV1
I7	~ION	Immediate
I8		Cases I8a, I8b, I8c
I8a	-	No effect on IIV1
I8b	ION & ISEL(IPID)	Immediate
I8c	~ION	Immediate
I9		Cases I9a and I9b
I9a	ION & IPID=nil	Immediate
I9b	ION & IPID=J & ISEL(J)	Immediate

The predicates that appear as post-conditions in the specifications are:

Predicate	Axiom(s)
~ION	P2, I7
Freeze=k & k<>0	P4, P7
Freeze=0	P6, P9
J0.ev	I1
RSCH	I3
REDO	I4

Operations on the I-machine can change only the following variables in the indicated manner.

Variable		Change
RSCH		Set to true (except I5)
REDO		Set to true (except I5)
ION		any
I.ev	for any I	any
I.st	for any I	any
P.st	for any P	Set to R or RW
IPID		any
RPID		any
AM		any

P-machine predicates

Non-interference with the P-machine predicates (ION, ~ION, Freeze=k) is established in section 8.5

I-machine predicates

RSCH and REDO can only by set on the I-machine. They are only reset when the P-machine again runs, the I-machine program having terminated.

Local Specifications

From the point of view of the P-machine, ION is invariant over all I-machine operations. Reason: if ION=false the I-machine cannot run so that ION is invariant. If ION=true then the I-machine can run, but the only transitions returning control to the P-machine (I5, I9) assert ION as

post-conditions.

Global Axiom	Local Axiom	Required Post-condition	Implication
P1	L1	.. & ION & ..	Immediate
P2	L2	~ION	Immediate
P3	L3	ION | ION | AM='I' & ..	(1)
P4	L4	Freeze=k+1	Immediate
P5	L5	Freeze=0	Immediate
P6	L6	Freeze=0 & ION	Immediate
P7	L7	Freeze=k-1	Immediate
P8	L8	.. & ION & ..	Immediate
P9	L9	Freeze=0 & ION	Immediate
P10	L10	.. & ION & ..	Immediate
P11	L10	.. & ION & ..	Immediate
I1-I4	IL1		Immediate
I5	IL2	AM='P'	I-program terminates
I6	IL3		Immediate
I7	IL5	~ION	Immediate
I8	IL4	ION | ...	(2)
I9	IL5	AM='P'	I-program terminates

(1) In case P3c, the argument above shows that ION must hold when control finally returns to the P-machine.

(2) In case I8c, control can only return to the interrupted process by I9 which has post-condition ION.

8.3 - PLZ/SYS Semantics

PLZ/SYS resembles PASCAL but is simpler. It includes arrays and records as in PASCAL and simple data types byte, word, short-integer, integer, and pointers to various types. The algorithms (see figure 8-2) are expressed in a subset of PLZ/SYS except for the replacement of pointers with implicit referencing and dereferencing.

The semantics of the control structures in the language are given in figure 8-1.

$$P^{e}_{x} \quad \{ x := e \} \quad P$$

$$\frac{P \ \{S\} \ P \ \& \text{ each EXIT statement in S has precondition Q, } \& \text{ each REPEAT statement in S has precondition P}}{P \ \{\text{do S od}\} \ Q}$$

$$\frac{B \ \& \ P \ \{S1\} \ Q \text{ and } \sim B \ \& \ P \ \{S2\} \ Q}{P \ \{\text{if B then S1 else S2 fi}\} \ Q}$$

$$\frac{B \ \& \ P \ \{S1\} \ Q \text{ and } (\sim B \ \& \ P) \Rightarrow Q}{P \ \{\text{if B then S1 fi}\} \ Q}$$

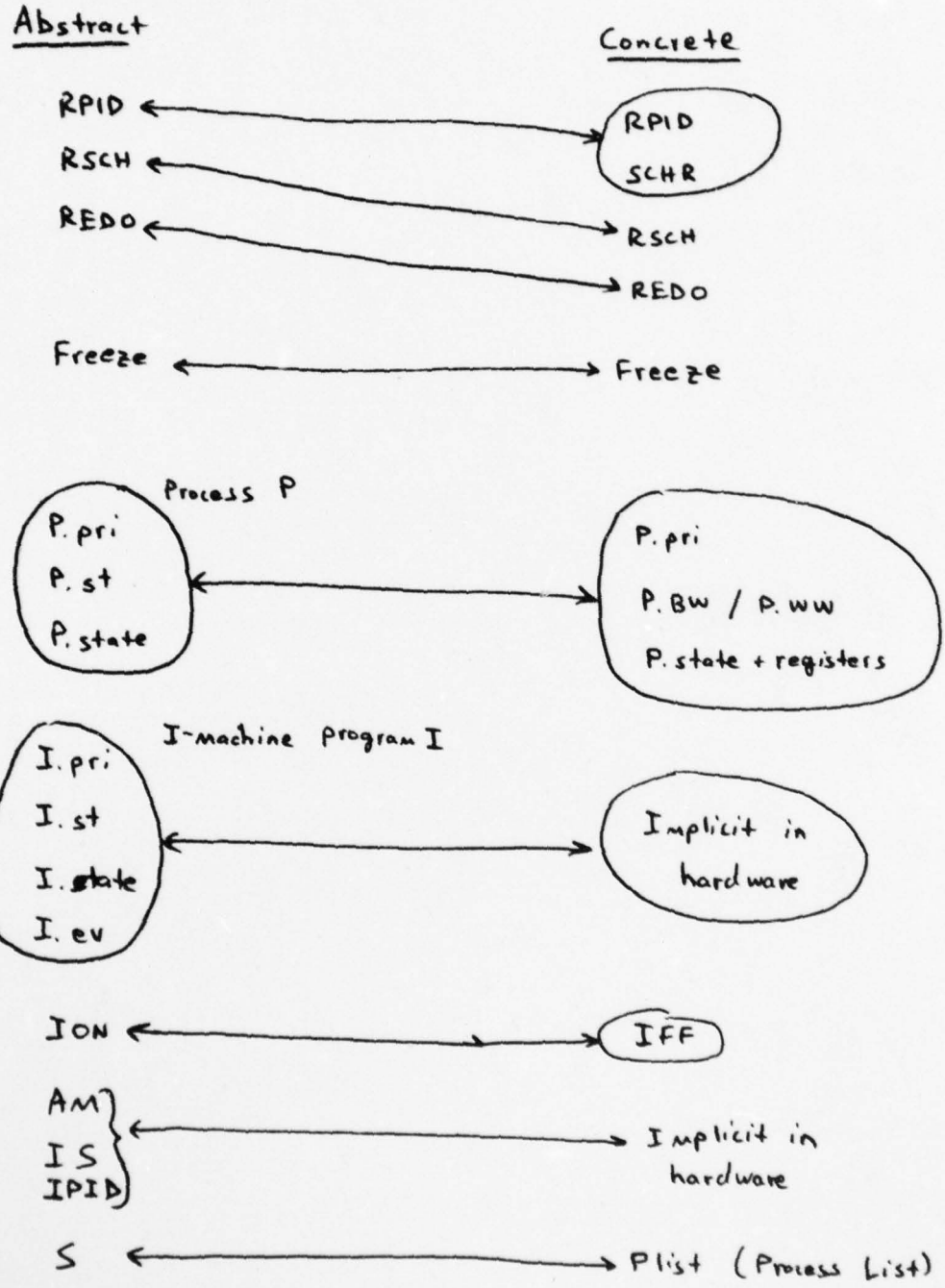
Figure 8-1. PLZ/SYS Axiomatic Semantics

8.4 - Abstract-Concrete Map

The Abstract-Concrete map relates each variable in the abstract representation to a variable in the concrete representation. Refer to section 7.3.2 for a discusstion of each element of the abstract representation.

Abstract	Concrete	Comments
RPID	RPID if SCHR=false nil if SCHR=true	4 bytes
IPID	implicit	implicit in hardware
Freeze	Freeze	1 byte
ION	IFF	a hardware register
RSCH	RSCH	1 byte
REDO	REDO	1 byte
P.pri	P.pri	1 byte (0..255)
P.st	P.BW and P.WW (boolean)	with the encoding: P.BW & ~P.WW <=> P.st=B ~P.BW & ~P.WW <=> P.st=R ~P.BW & P.WW <=> P.st=RW (P.BW & P.WW is an impossible state)
P.state	hardware registers	
I.pri	bus wiring	hardware determined
I.st	implicit	based on activation history which is hardware managed
I.state	hardware registers	
I.ev	hardware	
AM	implicit	hardware managed
S	Plist	Plist is a list ordered by priority.
IS	hardware	predetermined by system wiring

Figure 8.1. Concrete-Abstract map.



8.5 - Verification

Each of the operations must be verified to be consistent with the specifications. The predicates in the specifications are restated in terms of the concrete representation.

Only the instructions implemented in the Kernel software are verified. The instructions EI, DI, RETI, RESET, and INTR are implemented by the Z-80A hardware.

Each proof consists of two parts. First, the axiomatic derivation of the properties of the program text being examined is given. Then, arguments establishing safety and non-interference of each variable access and predicate are given.

Preliminary Discussion of Inteference

Some preliminary analysis of the specifications will simplify the non-interference arguments.

Pre-conditions for an involuntary context switch are:

RPID<>nil & Freeze=0

When these preconditions do not hold, the only variables

subject to interference are:

Variable	Axiom	Comments
P.st (P.WW, P.BW)	I1-I4	for any P in S
RSCH	I3	
REDO	I4	
I.ev	I6 and INTR	for any I<>J0
I.st	I8, I9, and INTR	
IPID	I8, I9, and INTR	

Any predicates not involving these predicates are interference-free.

Freeze=0 is not subject to interference. Reason: No I-machine operations can modify Freeze. If it is modified by the P-machine, it must be changed back to 0 before a context switch (CS) can cause the process that expected Freeze=0 to resume.

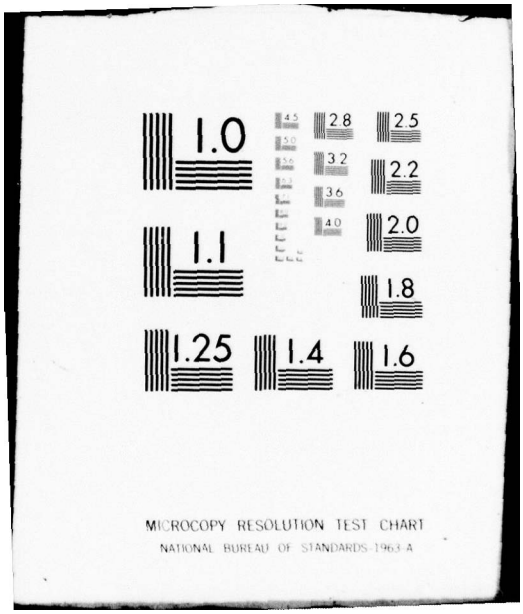
Freeze<>0 is not subject to interference. Reason: No I-machine operations can modity Freeze. Because Freeze<>0, no context switch can occur on the P-machine so that no operations that might modify Freeze can be executed.

ION is not subject to interference. Reason: ION=true is always restored by the I-machine before returning to an interrupted I- or P-machine process. If ION is modified by the P-machine, it is restored to ION=true as part of a context switch (CS) that resumes a process that had ION=true initially.

AD-A072 552 CALIFORNIA UNIV SANTA CRUZ INFORMATION SCIENCES F/G 9/2
THE DESIGN AND VERIFICATION OF AN OPERATING SYSTEM KERNEL.(U)
JUN 79 P G LEVY N00014-76-C-0682
UNCLASSIFIED TR-79-6-001 NL

2 OF 2
AD A072552

END
DATE FILMED
9-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

~ION is not subject to interference. Reason: No I-machine programs can run so that no operations that can affect ION can be executed by other than the running process.

Figure 8-2. Algorithms for the Kernel in terms of concrete representation.

```
FREEZE

        Freeze := Freeze+1

THAW

        if Freeze>0 then Freeze := Freeze-1;
                         if Freeze=0 and RSCH=true then CS fi
        fi

CS                              Local Variable: Best

        Save(RPID.state)
        SCHR := true; RPID := nil
        do
            SCHR := true; REDO := false
            Best := Plist

            do
                if Best=nil then exit fi
                if Best.BW=false then exit fi
                Best := Best.link
            od

            DI
            if REDO=true or Best=nil then EI; repeat
                                      else exit
            fi
        od

        RPID := Best;   SCHR := false;  RSCH := false
        EI
        Load(RPID.state)

WAKE (P)

        FREEZE; DI

        if P.BW=true then P.BW := false
        else if P.WW=false then P.WW := true fi
        fi

        EI
        if P.pri>RPID.pri then RSCH := true fi

        THAW
```

IWAKE (P) Local Variable: IONsave

```
IONSave := ION
DI

if P.BW=true then P.BW := false
else if P.WW=false then P.WW := true fi
fi

if IONSave=true then EI fi

if SCHR=true then REDO := true
             else if Freeze<>0 then RSCH:=true
                  else if P.pri>RPID.pri then ARMINT
                       fi
                  fi
fi
```

BLOCK (P)

```
FREEZE; DI
if P.WW=true then P.WW := false
          else if P.BW=false then P.BW := true fi
fi

EI

if RPID=P then Freeze := 0; CS
          else THAW
fi
```

ARMINT

```
INTR J0     { Force interrupt of J0 by setting J0.ev }
```

Load (state)

{Set the hardware state of the processor to state.}

Save (state)

{Save the hardware state of the processor in state.}

CS

Specification

```
P1      Freeze=0          {CS}    ION

        with oneof
P1a                       {CS}    PSEL(RPID) & ~RSCH & ~REDO
```

The predicate PSEL(P) restated in terms of the concrete representation is:

PSEL(P): ~P.BW & ∀ R: R.pri>P.pri => R.BW

Process list property:

P<>nil & P.link<>nil => P.pri>P.link.pri

Proof

```
1  {Freeze=0}

        Save(P.state)

2  {Freeze=0}

        SCHR := true; RPID := nil

3  {Freeze=0 & SCHR & RPID=nil }

        DO

4  {Freeze=0 & SCHR}

             SCHR := true; REDO := false
             Best := Plist

5  {Freeze=0 &
   SCHR &
    [ (REDO | (Best<>nil & ∀ R: R.pri>Best.pri => R.BW))
    | (REDO | (Best=nil  & ∀ R: R.BW))]  }

             DO

6  {Freeze=0 &
  SCHR &
    [ (REDO | (Best<>nil & ∀ R: R.pri>Best.pri => R.BW))
    | (REDO | (Best=nil  & ∀ R: R.BW))]  }
```

```
                if Best=nil then

7  {Freeze=0 &
   SCHR &
   (REDO | (Best=nil & ∀ R: R.BW))  }
                                    exit fi

8  {Freeze=0 &
   SCHR &
   (REDO | (Best<>nil & ∀ R: R.pri>Best.pri => R.BW)) }

                if Best.BW=false then

9  {Freeze=0 &
   SCHR &
   (REDO | (Best<>nil & ∀ R: R.pri>Best.pri => R.BW)) &
   ~Best.BW }
                                    exit fi

10 {Freeze=0 &
   SCHR &
   [(REDO|(Best.link<>nil & ∀ R: R.pri>Best.link.pri => R.BW)) |
    (REDO|(Best.link=nil  & ∀ R:R.BW)) ]}

                Best := Best.link

11 {Freeze=0 &
   SCHR &
   [(REDO | (Best<>nil & ∀ R: R.pri>Best.pri => R.BW)) |
    (REDO | (Best=nil  & ∀ R: R.BW)) ] }

             OD

12 {Freeze=0 &
   SCHR &
   [(REDO | (Best=nil  & ∀ R:R.BW )) |
    (REDO | (Best<>nil & PSEL(Best))) ] }

             DI

13 {Freeze=0 &
   SCHR &
   [(REDO | (Best=nil  & ∀ R:R.BW )) |
    (REDO | (Best<>nil & PSEL(Best))) ] &
   ~ION }

             if REDO=true or Best=nil then

14 {Freeze=0 &
```

```
  SCHR &
  ~ION }

                                          EI

15 {Freeze=0 &
  SCHR &
  ION }

                                        repeat
                                     else

16 {Freeze=0 & SCHR & ~REDO & Best<>nil & PSEL(Best) & ~ION }

                                        exit
            fi

17 {false}

        OD

18 {Freeze=0 & SCHR & ~REDO & Best<>nil & PSEL(Best) & ~ION }

        RPID := Best
        SCHR := False

19 {Freeze=0 & ~SCHR & ~REDO & RPID<>nil & PSEL(RPID) & ~ION }

        EI

20 {Freeze=0 & ION}

        Load(RPID.state)

21 {Freeze=0 & ION}
```

Comments

We must establish non-interference for each predicate and safe access for each variable.

Predicates 3, 4, 14, 15, and 17 are interference-free as they involve no variables that the I-machine can modify and the P-machine cannot context switch because SCHR=true.

In predicates 5-13, the variables Freeze, SCHR, and ION are not subject to interference. The clause (REDO | ...) is also interference-free because the I-machine can only set REDO=true thus making the clause true. It does this exactly

when variables P.st are modified (I4). Thus, if there is interference with P.st, REDO is set to true insuring that the clause remains true.

Predicates 16, 18, and 19 are interference-free as they include ~ION are a term. The I-machine cannot run if ~ION.

Finally, predicates 1, 2, 20, and 21 are locally interference-free.

Safe access is considered in the following table:

Variable	Access	Reason for Safety
SCHR	R/W	Atomic access
RPID	R/W	SCHR=true forbids other accessors
Best	R/W	Process local variable
REDO	R/W	Atomic access
P.BW	R	Atomic access
P.link	R	Read-only access, all accessors

The inner loop terminates because of the finite and unchanging length of the process list. The outer loop does not necessarily terminate. This is not an error, however, for if there is a ready process, the loop can terminate. If no interrupt programs execute IWAKE while the loop is running, it will terminate. If interrupt programs continually wake P-machine processes, the loop won't terminate.

Specification

P4 Freeze=k & k<100 {FREEZE} Freeze=k+1

Proof

1 {Freeze=k & Freeze<100}

```
Freeze := Freeze+1
```

2 {Freeze=k+1 & Freeze<=100}

Comments

Predicate	Reason for interference-free
Freeze=0	See beginning of section 8.5
Freeze<>0	See beginning of section 8.5

Variable	Reason for safe access
Freeze	Atomic access

THAW

Specification

P5 Freeze=0 {THAW}

Proof

1 {Freeze=0}

```
if Freeze>0 then Freeze := Freeze-1;
                 if Freeze=0 and RSCH=true then CS fi
fi
```

2 {Freeze=0}

Comments

Predicate	Reason for interference-free
Freeze=0	See beginning of section 8.5

Variable	Reason for safe access
Freeze	Atomic access

Specification

P6	Freeze=1	{THAW}	Freeze=0
P6a	with oneof	{THAW}	RPID=T & PSEL(T) & ~RSCH

Proof

```
1  {Freeze=1 & PIV2}
        if Freeze>0 then
2  {Freeze=1 & PIV2}
                        Freeze := Freeze-1;
3  {Freeze=0 & PIV2}
                         if Freeze=0 and RSCH=true then
4  {Freeze=0}
                                  CS
5  {Freeze=0 & ION} end of proof for this path.
                         fi
6  {Freeze=0 & PIV2}
        fi
7  {Freeze=0 & PIV2}
```

Comments

Predicate	Reason for interference-free
Freeze<>0	See beginning of section 8.5
Freeze=1 & PIV2	Freeze=1 is locally interference-free and PIV2 is invariant, hence locally interference-free.
PIV2	Locally interference-free

Variable	Reason for safe access
Freeze	Atomic access
RSCH	Atomic access

Specification

P7 Freeze=k & k>1 {THAW} Freeze=k-1

Proof

1 {Freeze=k & k>1}

if Freeze>0 then

2 {Freeze=k & k>1}

Freeze := Freeze-1;

3 {Freeze=k-1 & Freeze>0}

if Freeze=0 and RSCH=true then CS fi

fi

4 {Freeze=k-1}

Comments

See P6 comments

BLOCK (P)

Specification

P8	Q<>RPID	{BLOCK Q}	ION
	with oneof		
P8a	Q.st=B | Q.st=R	{BLOCK Q}	Q.st=B
P8b	Q.st=RW	{BLOCK Q}	Q.st=R

Proof

1 {Q<>RPID & Freeze=k}

FREEZE; DI

2 {Q<>RPID & Freeze=k+1 & ~ION}

```
        if P.WW=true then
3               {~P.BW & P.WW} P.WW:=false {~P.BW & ~P.WW}
                   else if P.BW=false then
4               {~P.BW & ~P.WW} P.BW:=true fi {P.BW & ~P.WW}
        fi
        EI
5   {Q<>RPID & Freeze=k+1 & ION}
        if RPID=Q then Freeze := 0; CS
                  else
6   {Q<>RPID & Freeze=k+1 & ION}
                        THAW
7   {Q<>RPID & Freeze=k & ION}
        fi
8   {Q<>RPID & Freeze=k & ION}
```

Comments

Predicate	Reason for interference-free
Freeze<>0	See beginning of section 8.5
Freeze=k	See beginning of section 8.5
ION	See beginning of section 8.5
~ION	See beginning of section 8.5
3, 4	~ION & Freeze>0 so full exclusion allows sequential proof.

Variable	Reason for safe access
P.WW, P.BW	All context switching disabled
RPID	Freeze>0 prevents any write access

Specification

```
P9      Q=RPID          {BLOCK Q}       Freeze=0 & ION
        with oneof
```

```
P9a     Q.st=B | Q.st=R {BLOCK Q}   Q.st=B & RPID=T & PSEL(T)
P9b     Q.st=RW         {BLOCK Q}   Q.st=R & RPID=T & PSEL(T)
```

Proof

```
1  {Q=RPID & Freeze=k}

        FREEZE; DI

2  {Q=RPID & Freeze=k+1 & ~ION}

        if P.WW=true then P.WW := false
                  else if P.BW=false then P.BW := true fi
        fi

        EI

3  {Q=RPID & Freeze=k+1 & ION}

        if RPID=P then

4  {Q=RPID & Freeze=k+1 & ION}

                        Freeze := 0;

5  {Q=RPID & Freeze=0 & ION}

                        CS

6  {Freeze=0 & ION}  end of this path

                  else THAW
        fi
```

Comments

See P9, above.

WAKE (P)

Specification

```
P10     Freeze=0        {WAKE Q}        ION

        with oneof

P10a    Q.st=R | Q.st=RW {WAKE Q}       Q.st=RW
```

p10b Q.st=B {WAKE Q} Q.st=R

Proof

```
1  {Freeze=0 & PIV2}

        FREEZE; DI

2  {Freeze=1 & PIV2 & ~ION}

        if Q.BW=true then

3           {Q.BW & ~Q.WW} Q.BW := false {~Q.BW & ~Q.WW}

        else if Q.WW=false then

4           {~Q.BW & ~Q.WW} Q.WW := true {~Q.BW & Q.WW}
            fi
        fi

        EI

5  {Freeze=1 &
  ION &
  ~RSCH => (∀ R<>Q: R.pri > RPID.pri => R.BW) }

        if Q.pri>RPID.pri then

6  {Freeze=1 & ION & Q.pri>RPID.pri}

                    RSCH := true
7  {Freeze=1 & ION & PIV2}

        fi

8  {Freeze=1 & ION & PIV2}

        THAW

9  {Freeze=0 & ION}
```

Comments

Predicate	Reason for interference-free
Freeze=k	See beginning of section 8.5
PIV2	invariant, so interference-free
~RSCH => (∀ R<>Q: R.pri > RPID.pri => R.BW)	Same argument as PIV2
ION	See beginning of section 8.5

~ION	See beginning of section 8.5
3, 4	~ION & Freeze>0 so full exclusion allows sequential proof.

Variable	Reason for safe access
Q.WW, Q.BW	All context switching disabled
RPID	Freeze>0 prevents any write access
RPID.pri	Atomic access (given safe access to RPID)

Specification

P11	Freeze<>0	{WAKE Q}	ION & ~RSCH => (RPID=T & PSEL(T))
	with oneof		
P11a	Q.st=R | Q.st=RW	{WAKE Q}	Q.st=RW
p11b	Q.st=B	{WAKE Q}	Q.st=R

Proof

```
1  {Freeze=k & k>0}

        FREEZE; DI

2  {Freeze=k+1 & k>0 & ~ION}

        if Q.BW=true then

3               {Q.BW & ~Q.WW} Q.BW := false {~Q.BW & ~Q.WW}

        else if Q.WW=false then

4               {~Q.BW & ~Q.WW} Q.WW := true {~Q.BW & Q.WW}
            fi
        fi

        EI

5  {Freeze=k+1 & k>0 & ION &
   (~RSCH => (∀ R<>Q: R.pri>RPID.pri=>R.BW)) }

        if Q.pri>RPID.pri then

6  {Freeze=k+1 & k>0 & ION &
   Q.pri>RPID.pri &
   ~Q.BW }

                                RSCH := true
```

```
7  {Freeze=k+1 & k>0 & ION &
   (~RSCH => (∀ R: R.pri>RPID.pri => R.BW)) }

        fi

8  {Freeze=k+1 & k>0 & ION &
   (~RSCH => (∀ R: R.pri>RPID.pri => R.BW)) }

        THAW

9  {Freeze=k & k>0 & ION & PIV2}
```

Comments

Predicate	Reason for interference-free
Freeze=k	See beginning of section 8.5
PIV2	invariant, so interference-free
~RSCH => (∀ R<>Q: R.pri > RPID.pri => R.BW)	Same argument as PIV2
ION	See beginning of section 8.5
~ION	See beginning of section 8.5
3, 4	~ION & Freeze>0 so full exclusion allows sequential proof.
~Q.BW	Freeze>0 so no P-machine program can change Q.BW, and there are no I-machine operations that can set Q.BW=true.

Variable	Reason for safe access
Q.WW, Q.BW	All context switching disabled
RPID	Freeze>0 prevents any write access
RPID.pri	Atomic access (given safe access to RPID)

IWAKE (P)

Specification

```
I1      Freeze=0 &          {IWAKE Q}
        RPID<>nil &
        Q.pri>RPID.pri

        with oneof

I1a     Q.st=R | Q.st=RW {IWAKE Q}      Q.st=RW & J0.ev
I1b     Q.st=B           {IWAKE Q}      Q.st=R  & J0.ev
```

```
I2      Freeze=0 &          {IWAKE Q}
        RPID<>nil &
        Q.pri<=RPID.pri

        with oneof

I2a     Q.st=R | Q.st=RW    {IWAKE Q}       Q.st=RW
I2b     Q.st=B              {IWAKE Q}       Q.st=R

I3      Freeze<>0 &         {IWAKE Q}       RSCH
        RPID<>nil

        with oneof

I3a     Q.st=R | Q.st=RW    {IWAKE Q}       Q.st=RW
I3b     Q.st=B              {IWAKE Q}       Q.st=R

I4      RPID=nil            {IWAKE Q}       REDO

        with oneof

I4a     Q.st=R | Q.st=RW    {IWAKE Q}       Q.st=RW
I4b     Q.st=B              {IWAKE Q}       Q.st=R
```

The proof of the first few statements is common to all parts.

Proof

```
1 {ION'=ION}

        IONSave := ION

2 { IONSave=ION' }

        DI

3 { IONSave=ION' & ~ION }

        if Q.BW=true then

4               {Q.BW & ~Q.WW} Q.BW := false {~Q.BW & ~Q.WW}

        else if Q.WW=false then

5               {~Q.BW & ~Q.WW} Q.WW := true {~Q.BW & Q.WW}
            fi
        fi

6 { IONSave=ION' & ~ION }
```

```
        if IONSave=true then
7 {ION'=true & ~ION }
                                    EI fi
8 { ION=ION' }


                if SCHR=true then
9 { SCHR } case I4
                                        REDO := true
10 { SCHR & REDO }  end of case I4
                                  else if Freeze<>0 then
11 { ~SCHR & Freeze<>0 }  case I3
                                              RSCH:=true
12 { ~SCHR & Freeze<>0 & RSCH } end case I3
                                  else
13 { ~SCHR & Freeze=0 }
                                       if Q.pri>RPID.pri then
14 { ~SCHR & Freeze=0 & Q.pri>RPID.pri } case I1
                                                  ARMINT
15 { J0.ev } end case I1
                                     fi
                                  fi
                fi
```

Comments

Predicate	Reason for interference-free
Freeze=0	Writer is lower priority
Freeze<>0	Writer is lower priority
ION	See beginning of section 8.5
~ION	See beginning of section 8.5
IONSave	Exclusive access to IONSave
4, 5	~ION so full exclusion

	allows sequential proof.
SCHR, ~SCHR	Writer is lower priority
REDO	Processes that can set REDO=false are lower priority
RSCH	Processes that can set RSCH=false are lower priority
Q.pri>RPID.pri	Processes that affects these variables are lower priority
J0.ev	Process that can set J0.ev=false is lower priority

Variable	Reason for safe access
Q.WW, Q.BW	All context switching disabled
SCHR	Atomic access
RSCH	Atomic access
REDO	Atomic access
IONSave	Exclusive access
RPID	Writer is lower priority
RPID.pri	Atomic access (given safe access to RPID)
Q.pri	Atomic access

Specification

I5	IPID=J0	{RETICS}	
	with oneof		
I5a	Freeze=0 & (forall K<>J0: K.st=idle)	{RETICS}	IPID=nil & J0.st=idle & AM='P' & RPID=T & PSEL(T) & ~RSCH & ~REDO & ION

Proof

1 {Freeze=0}

CS

2 {ION}

The oneof predicates are taken care of by the execution of CS and the Z-80A hardware.

8.6 - Verification of Theorems

Theorem 1. If a BLOCK P is executed, P will not run until a WAKE P (or IWAKE P) is executed.

Proof

Suppose the first oneof axiom (P8a or P9a) applies. Then P.st=B and P cannot run because PSEL(P) cannot be true as it must (P1a) for P to continue execution. The only transitions that can change P.st=B are P10b, P11b, I1b, I2b, I3b, and I4b. Each of these is a WAKE or IWAKE operation.

In the case of the second oneof axiom (P8b and P9b), the pre-condition P.st=RW implies that a WAKE P or IWAKE P has already been executed as WAKE and IWAKE are the only operations that have P.st=RW as post-conditions.

Theorem 2. If no WAKE P or IWAKE P operations are executed, a process P will not run after at most 2 BLOCK P operations are executed.

Proof

Suppose P.st=RW. Then after one execution of BLOCK P, P.st=R (P8b, P9b). BLOCK, WAKE, and IWAKE are the only operations affecting P.st and, by hypothesis, WAKE and IWAKE are not executed. In this case, or if P.st=R already, an execution of BLOCK P results in P.st=B (P8a, P9a). P cannot run when P.st=B. Thus, at most 2 executions of BLOCK P with none of WAKE P results in P.st=B.

Theorem 3. A P-machine context switch will not be performed unless Freeze=0.

Proof

Follows directly from P1 and I5 as Freeze=0 is a pre-condition for both of these operations and no other operations cause context switching on the P-machine.

This completes the verification of the Kernel.

9.1 - Review of Results

We have discussed the value of structure to program design, implementation, and verification. When a system is designed as a hierarchy of virtual machines, each of the levels can be independently specified and verified. Abstraction provides similar engineering benefits by facilitating the decomposition of the task. Verification can again be decomposed along the abstraction lines.

Parallelism significantly complicates the verification task by exposing the internal states in the implementation of abstractions and virtual machines. Programs running in parallel with users of abstractions can observe intermediate states of variables that the abstractions alter.

Reducing the number and subtlety of the interconnections in a program tends to simplify the implementation, maintenance, and verification tasks. Implementation is simplified by allowing several people to work simultaneously on a program with minimal interaction. Maintenance is facilitated by minimizing the scope of changes made to a prgram to a subset of that program. Verification is simplified because, even with parallelism, the amount of the program that must be

examined to establish some property of a part of it is minimized.

Abstraction and virtual machines also simplify the specification task. The individual abstractions and virtual machines can be specified individually. Additional, more global, specifications relate them.

The idea of virtual machines and interrupt disabling has been combined in a new technique for mutual exclusion. If context switching among a group of processes is disabled, then the one of them that is running has exclusive access to all its variables. None of the other processes can interfere until context switching is again enabled.

Real-time responsiveness can be maintained by establishing a separate virtual machine on which all but interrupt programs run. When context switching is disabled on this virtual machine, interrupt servicing is unaffected. Mutual exclusion among all non-interrupt processes is achieved.

This technique is easy to implement, requires no queues or other complex data structures, and has positive effect on the verification task. When context switching is disabled, only a single process can execute so that sequential

verification techniques can be applied.

Standard axiomatic specification techniques were demonstrated to be inadequate for the specification of operations that run in a parallel environment with shared variables. The axiomatic specification technique was extended so that each axiom includes pre- and post-conditions that can be used as assertions in a proof, and also pre- and post-conditions that characterize some action that the operation performs. These latter assertions cannot be used in proofs, nor can they be assumed true even if execution of the operation resulted in them being true momentarily. These assertions are used to establish more global properties of the system.

The use of predicates to describe states of programs and in parallel program proofs was explored. Use of predicates in parallel programs must be restricted so that the truth of the predicates cannot change as a result of action by other processes. Several conditions sufficient to assure this have been presented.

We restricted our attention to a small region of a program that we call a basic computation. The basic computation contains no instructions that voluntarily release the CPU

for use by other processes. Predicates used in the verification of basic computations will not change if the basic computation is a region that excludes from execution all processes that can affect variables in any of the predicates. This condition covers the cases where there are no other processes that can affect variables as well as the case where context switching is disabled so that no other processes can run.

Priority based scheduling can provide an environment that assures that certain variables will be unchanged during the execution of a basic computation. If all processes that read (but not write) the particular variables are of higher priority than processes that modify the variables, then any actions based on examination of such shared variables in the same basic computation cannot be affected by processes that modify the variables. This situation occurs when interrupt programs examine variables describing the state of the system that are set by normal, lower priority, processes.

Conditions that assure that variables that can be accessed atomically (set and loaded properly) were also examined. Sufficient conditions include true (machine dependent) atomic access, exclusion among accessors, access by only a single process, and read-only access by all processes.

Determination and verification of the state of accessors of a variable might be necessary to conclude that access to a variable yields a defined value if none of the other conditions apply.

An operating system kernel (called the Kernel) has been designed and specified. The Kernel provides a multiprogramming environment on a Z-80A microprocessor. The specifications consist of an abstract representation that provides a model of the Kernel, a set of virtual machines with different instruction sets that programs can run on, extended axioms that characterize each operation on each virtual machine, axioms that describe the operations from the point of view of a process that executes them, invariants that describe global properties of the system, and theorems the state global properties of some operations.

Finally the Kernel was verified. The verification consists of several parts. The local and global specifications of the operations were shown to be consistent. The invariants were shown to be invariant over each of the operations on each virtual machine. The semantics of the language PLZ/SYS (in which the Kernel is written) was given. A mapping between the abstract representation and the concrete representation used in the implementation was described.

Next the algorithms were shown to be consistent with the specifications. This involved both a derivation similar to conventional axiomatic proofs and arguments that the predicates used are not modified by any other processes. The theorems were proved by considering possible actions performed by some of the operations that could not be carried in predicates because of interference from other processes.

9.2 - Value of Verification Analysis

This thesis claims to address the practical aspects of operating system design and verification; some comments on the value and difficulty of the task is in order. As noted in the introduction, program correctness is important but only one of several valuable program properties. We conclude, however, that structuring that contributes to engineering aspects of a program is not in conflict with structure that aids verification. The practical value of good engineering properties are clear in reducing cost and increasing reliability and applicability of a program.

The practicality of performing an extensive verification is

questionable. We do, however, claim that some of the analysis that underlies the proof is valuable and practical. Many of the complex problems that appear in real-time systems are related to variable access or modification in an unexpected state. In the course of the verification of the Kernel, one error of this nature was uncovered.

The criteria given for safe variable access can be applied to all variables in the program in a fairly mechanical way. Potential trouble spots can thus be identified. The analysis required for proving the soundness property of basic computations is also valuable in identifying potential trouble spots in a system. The analysis identifies what is assumed by various computations in the program and where in the program those can be affected. It is likely that these analyses would uncover most serious and subtle errors in a system. This type of error is also unlikely to be revealed in testing as subtle issues of timing and sequencing may be involved. The training required to analyse programs for these problems is also not extensive.

9.3 - Difficulty of Specifications

The development of the specifications was of similar complexity to the proof itself. This is partially due to

RSCH := true

the continual refinements that were required as the specification techniques themselves were refined. Experience has been beneficial in making the specification production easier.

Still, the complexity and difficulty of finding an appropriate model and axiomatizing the operations is not to be underestimated. Analysis of what variables will be consistently accessable is not an issue in the specifications, but identification of those usable in predicates in proofs is.

Subtle errors in the specifications of the P and I-machine operations were found at a uniform rate even as the thesis approached completion. There is no reason to believe that no errors remain; nor is there any obvious way to get to such a conviction.

9.4 - Further Research

There are many directions for further research.

- Much of the technique exposed in this thesis is left informal. More formal and perhaps mechanical

techniques can be developed.

- It seems likely that the detection of unsafe access to variables can be done mechanically. What properties of a language are required to do this?

- Why are specifications so hard to write? How can they be produced more easily? (Additional experience might help.)

- Can a weakest-precondition like calculus can be defined that expresses the soundness property of basic computations in terms of the structure of the language in which the program is written?

- What other criteria are there for safety and soundness? Can they be mechanically checkable?

- Can the notion of binding be formalized? Would a measure of it help in evaluating quality of software and language design?

- How can time be added to specifications?

[Brinch-Hansen 73] P. Brinch-Hansen Operating System Principles Prentice-Hall (July 1973)

[Brinch-Hansen 76] P. Brinch-Hansen "The Programming Language Concurrent PASCAL" in Language Hierarchies and Interfaces Springer-Verlag (1976)

[Clint & Hoare 72] M. Clint, C. A. R. Hoare "Program Proving: Jumps and Functions" Acta Informatica 1, 214-224, (1972)

[Denning 74] P. J. Denning "Is Structured Programming Any Longer the Right Term?" SIGPLAN Notices, (November 1974)

[DeRemer & Levy 79] F. DeRemer, P. Levy "Summary of the Characteristics of Several 'Modern' Programming Languages" SIGPLAN Notices, (May 1979)

[Dijkstra 68] E. W. Dijkstra "The Structure of the "THE"-Multiprogramming System" CACM 11, 5 (May 1968)

[Dijkstra 72] E. W. Dijkstra "Structured Programming" in Structured Programming Academic Press, (1972)

[Flon 77] L. Flon "On the Design and Verification of Operating Systems" Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, (May 1977)

[Floyd 67] R. W. Floyd "Assigning Meanings to Programs" Proc. of Symposia in Applied Math., (1967)

[Gries 76] D. Gries "An Exercise in Proving Parallel Programs Correct" in Language Hierarchies and Interfaces, Springer Verlag (1976)

[Habermann et al 76] A. N. Habermann, L. Flon, L. Cooprider "Modularization and Hierarchy in a Family of Operating Systems" CACM 19, 5 (May 1976)

[Hansen 70] P. B. Hansen "The Nucleus of a Multiprogramming System" CACM 13, 4 (April 1970)

[Hoare 69] C. A. R. Hoare "An Axiomatic Basis for Computer Programming" CACM 12, 10 (October 1969)

[Hoare 71a] C. A. R. Hoare "Procedures and Parameters: An Axiomatic Approach" in Symposium on Semantics of Algorithmic Languages Springer Verlag (1971)

[Hoare 73] C. A. R. Hoare, N. Wirth "An Axiomatic Definition of the Programming Language PASCAL" Acta Informatica 2 (1973)

[Hoare 74] C. A. R. Hoare "Monitors: An Operatinf System Structuring Concept" CACM 17, 10 (October 1974)

[Horning & Randell 73] J. J. Horning, B. Randell "Process Structuring" ACM Computing Surveys 5, 1 (March 1973)

[Jensen 75] K. Jensen, N. Wirth PASCAL User Manual and Report Springer-Verlag (1975)

[Liskov 72] B. H. Liskov "A Design Methodology for Reliable Software Systems" Proc. FJCC, (1972)

[Liskov & Zilles 75] B. H. Liskov, S. N. Zilles "Specification Techniques for Data Abstractions" IEEE Transactions on Software Engineering, (May 1975)

[London et al 77] R. London, et al "Proof Rules for the Programming Language EUCLID" USC Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90291

[Morgan 77] D. E. Morgan, D. J. Taylor "A Survey of Methods of Achieving Reliable Software" IEEE Computer (February 1977)

[Myers 73] G. J. Myers "Composite Design: The Design of Modular Programs" IBM Systems Development Division, (January 1973)

[Owicki & Gries 76a] S. Owicki, D. Gries "Verifying Properties of Parallel Programs: An Axiomatic Approach" CACM 19,5 (May 1976)

[Owicki & Gries 76b] S. Owicki, D. Gries "An Axiomatic Technique for Parallel Programs I" Acta Informatica 6, 319-340 (1976)

[Parnas 71] D. L. Parnas "Information Distribution Aspects of Design Methodology" IFIPS 1971

[Parnas 72a] D. L. Parnas "A Technique for Software Module Specifications with Examples" CACM 15, 5, (May 1972)

[Parnas 72c] D. L. Parnas "On the Criteria To Be Used in Decomposing Systems into Modules" CACM 15, 12, (December 1972)

[Parnas 75b] D. L. Parnas "The Influence of Software Structure on Reliability" SIGPLAN Notices (June 1975)

[Parnas 75c] D. L. Parnas "Use of the Concept of Transparency in the Design of Hierarchically Structured Systems" CACM 18, 7, (July 1975)

[Parnas 75d] D. L. Parnas, G. Handzel "More Specification Techniques for Software Modules" Tech. Rep. Fachbereich Informatik, (1975)

[Parnas 76a] D. L. Parnas, H. Wurges "Response to Undesired Events in Software Systems" 2nd Int. Conf. on Software Engineering, (October 1976)

[Parnas 76b] D. L. Parnas "On the Design and Development of Program Families" IEEE Transactions on Software Engineering (March 1976)

[Parnas 76c] D. L. Parnas, G. Handzel, H. Wurges "Design and Specification of the Minimal Subset of an Operating System Family" IEEE Transactions on Software Engineering, (December 1976)

[Parnas 76d] D. L. Parnas "Some Hypotheses about the 'Uses' Hierarchy for Operating Systems" Tech. Rep., Fachbereich Informatik, (January 1976)

[Parnas 77] D. L. Parnas "Designing Software for Ease of Extension and Contraction" in Proceedings of the 3rd International Conference on Software Engineering (1977)

[Saxena 76] A. Saxena "A Verified Specification of a Hierarchical Operating System" Technical report 107, Stanford Electronics Lab., Stanford, California

[Scott 71] D. Scott, C. Strachey "Toward a Mathematical Semantics for Computer Languages" Proc. of Symposium on Comp. and Automata, (April 1971)

[Snook et al 78] T. Snook, et al "Report on the Programming Language PLZ/SYS" Springer-Verlag (1978)

[Weissman 74] L. M. Weissman "A Methodology for Studying the Psychological Complexity of Computer Programs" Tech. Rep. CSRG-37, (August 1974), Univ. of Toronto

[Wirth 71] N. Wirth "Program Development by Stepwise Refinement" CACM 14, 4, (April 1971)

[Zilog 77] Zilog, Inc. Z-80 CPU Technical Manual Zilog, Inc., 10460 Bubb Rd., Cupertino, CA, 95014

OFFICIAL DISTRIBUTION LIST

Contract N00014-76-C-0682

Defense Documentation Center
Cameron Station
Alexandria, VA 22314
12 Copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 Copies

Office of Naval Research
Code 200
Arlington, VA 22217
1 Copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 Copy

Office of Naval Research
Branch Office, Boston
Bldg. 114, Section D
666 Summer Street
Boston, MA 02210
1 Copy

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, ILL 60605
1 Copy

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106
1 Copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375
6 Copies

Dr. A.L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps (Code RD-1)
Washington, D.C. 20380
1 Copy

Naval Ocean Systems Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152
1 Copy

Mr. E.H. Gleissner
Naval Ship Research & Development Center
Computation and Mathematics Department
Bethesda, MD 20084
1 Copy

Captain Grace M. Hopper (008)
Naval Data Automation Command
Washington Navy Yard
Building 166
Washington, D.C. 20374
1 Copy

Mr. Glick (R53)
Director
National Security Agency
Fort G.G.
Meade, MD 20755
1 Copy