AD-A040 699 STANFORD UNIV CALIF STANFORD ELECTRONICS LABS F/G 9/2
A VERIFIED SPECIFICATION OF A HIERARCHICAL OPERATING SYSTEM.(U)
JAN 76 A R SAXENA N00014-75-C-0601
UNCLASSIFIED TR-107 NL

1 OF 3
AD A040699



AD A040699

SEL-76-011

A Verified Specification of a Hierarchical Operating System

by

Ashok R. Saxena

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

January 1976

Reproduction in whole or in part is permitted for any purpose of the United States Government.

DDC
RECEIVED
JUN 17 1977
A

Technical Report No. 107

This research was supported by the National Science Foundation under Grant number GJ41644 and by the Joint Services Electronics Program, U.S. Army, U.S. Navy, and U.S. Air Force under contract number N0014-75-C-0601.

DIGITAL SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES

STANFORD UNIVERSITY · STANFORD, CALIFORNIA

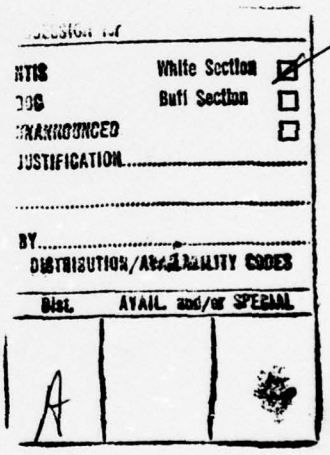
AD No.
DDC FILE COPY

SU-SEL-76-011

A VERIFIED SPECIFICATION OF A HIERARCHICAL OPERATING SYSTEM

by

Ashok R. Saxena



January 1976

Technical Report No. 107

DIGITAL SYSTEMS LABORATORY
Department of Electrical Engineering
Stanford University
Stanford, California



A VERIFIED SPECIFICATION OF A HIERARCHICAL OPERATING SYSTEM

by

Ashok R. Saxena

Technical Report No. 107

January 1976

Digital Systems Laboratory
Department of Electrical Engineering
Stanford University
Stanford, California

ABSTRACT

This thesis discusses the design, specification, and vertification of computer operating systems. The operating system problem considered, the many-process problem, is the design of an operating system that can support a large number of concurrent processes. This design problem is a vehicle to investigate the use of a design methodology, the hierarchical levels of abstraction methodology; the use of structured programming techniques in the specification of the system; and the development of techniques for the verification of concurrent programs, particularly operating system programs.

A solution to the many-process problem is obtained and it is shown that the hierarchical levels of abstraction methodology simplifies the conception of the solution and helps avoid potential deadlocks in the system. A PASCAL specification of the four levels of the system is given demonstrating the usefulness of structured programming techniques for specifying operating system programs. A detailed description of the development of the simple memory

manager, a complex and large segment of the system, is given to show the use of step-wise refinement for improving the efficiency of the program and as an aid in understanding its final specification. The specifications for the first two levels: simple scheduler and simple memory manager, are formally verified. The notion of exclusive access of a resource has been formalized and used in the verification of concurrent program. Sufficient conditions for verifying the absence of deadlocks in a system of monitors are also developed.

TABLE OF CONTENTS

CHAPTER		PAGE
1	INTRODUCTION	1
2	DESIGN	6
	2.1 HIERARCHICAL LEVELS OF ABSTRACTION	6
	2.2 THE MANY-PROCESS PROBLEM	10
	2.3 SOLUTION 1	11
	2.4 THREE-LEVEL SOLUTIONS	16
	2.5 FOUR-LEVEL SOLUTION	20
	2.6 A NON-HIERARCHICAL SOLUTION (with potential deadlock)	22
	2.7 CONCLUSIONS	25
3	SPECIFICATION	27
	3.1 STRUCTURED PROGRAMMING	28
	3.2 THE MONITOR CONCEPT	30
	3.3 SIMPLE SCHEDULER	33
	3.4 SIMPLE MEMORY MANAGER	47
	3.5 SCHEDULER	112
	3.6 MEMORY MANAGER	123
	3.7 CONCLUSION	126

CHAPTER PAGE

4 VERIFICATION 132

4.1 METHODOLOGY FOR VERIFICATION 132

4.2 VERIFICATION OF CONCURRENT PROGRAMS . . . 133

4.3 VERIFICATION OF THE SIMPLE SCHEDULER . . 152

4.4 VERIFICATION OF THE SIMPLE MEMORY MANAGER 156

5 CONCLUSION 159

APPENDIX A VERIFICATION RULES 162

APPENDIX B VERIFICATION OF THE SIMPLE SCHEDULER SPECIFICATION 169

B1 THE ABSTRACT DATA TYPE DISTINCT ELEMENT QUEUE 169

B2 VERIFICATION OF THE SIMPLE SCHEDULER PROGRAMS 179

B3 PROOF OF CORRECTNESS OF THE SIMPLE SCHEDULER THEOREMS 186

APPENDIX C VERIFICATION OF THE SIMPLE MEMORY MANAGER SPECIFICATION 195

C1 PARTIAL VERIFICATION OF THE SIMPLE MEMORY MANAGER SPECIFICATIONS . . . 198

CHAPTER PAGE

C2 PROOF OF TERMINATION AND ABSENCE OF DEADLOCKS 222

BIBLIOGRAPHY 238

LIST OF FIGURES

FIGURE		PAGE
2.1	A Non-hierarchical Solution	21
3.1	Monitor Declaration	31
3.2	Simple Scheduler Specification	42
3.3	Structure of the Simple Scheduler	46
3.4	Virtual-memory Version 0	50
3.5	Virtual-memory Version 1	56
3.6	Virtual-memory Version 2	64
3.7	Virtual-memory Version 3	70
3.8	Virtual-memory Version 4	78
3.9	Virtual-memory Version 5	82
3.10	Virtual-memory Version 6	89
3.11	Virtual-memory Version 7	95
3.12	Virtual-memory Version 8	104
3.13	Structure of the Simple Memory Manager	111
3.14	Scheduler Specifications	118
3.15	Structure of the Scheduler	124
3.17	Structure of the Memory Manager	128
3.18	Structure of the Four-level Solution	131

FIGURE PAGE

B.1 Implementation of Type T (distinct element queue) 174

B.2 Verification of the Simple Scheduler 181

C.1 Partial Verification of the Simple Memory Manager 199

C.2 Verification of Deadlock Invariants 226

CHAPTER 1. INTRODUCTION

This thesis discusses the design, specification, and verification of an operating system. The operating system problem considered, the many-process problem, is the design of an operating system that can support a large number of concurrent processes. This design problem is a vehicle to investigate the use of a design methodology, the hierarchical levels of abstraction methodology; the use of structured programming techniques in the specification of the system; and the development of techniques for the verification of concurrent programs, particularly operating system programs.

We have chosen the many-process problem because it is an interesting operating system problem. There are very few systems today that provide the facility for dynamic creation and destruction of processes: MULTICS [Organick 1972] and OS-MVT [Katzan 1973] are examples of such systems. However, these systems limit the number of concurrent processes that can exist by limiting the main memory space devoted to storing the process states. In the many-process

problem the number of processes is assumed to be so large that it is considered economically unfeasible to store all the process states permanently in the main memory, thus secondary memory is used for maintaining their states. Systems that can support a large number of concurrent processes, such as the solution to the many-process problem, are useful when there is a need for a large number of asynchronous processes; or when it is conceptually simpler to model solutions to programming problems as a large number of asynchronous processes. Examples of problems requiring a large number of asynchronous processes are the weather forecasting and transaction oriented systems.

Our motivation for investigating the hierarchical levels of abstraction methodology is that it has been suggested as a technique for dealing with the complexity of operating system design and as an aid to proving system correctness [Dijkstra 1968a, 1968b and Neumann et al. 1975]. Two well-known examples of the use of the methodology are the THE system [Dijkstra 1968b] and the VENUS system [Liskov 1972]. Both of these systems are small systems, that is, they support a small number of concurrent processes and these processes exist for the life-time of the system. In the many-process problem there is an apparent need for

violating the hierarchical restriction of the methodology. In this thesis we have shown how the methodology may be applied to overcome the apparent violation.

Our motivation for investigating the use of structured programming techniques in the specification is that it has been suggested as a means of making programs more easily understandable and verifiable. The concepts of structured programming are best illustrated by the examples in the book by Dahl, Dijkstra, and Hoare [Dahl, Dijkstra and Hoare 1972] and an article by Knuth [Knuth 1974]. However, the examples in the literature illustrating the use of structured programming are mostly small sequential programs, whereas their use is claimed to be critical in the design of large programs. It is one of the goals of this thesis to demonstrate their use in the specification of large programs such as the solution to the many-process problem.

Another major goal of this thesis is the verification of the specifications. Since the publication of Floyd's paper on the inductive assertion approach to the verification of programs [Floyd 1967], many people have worked on verifying programs either by hand [Hoare 1971b, London 1970] or by building systems to verify them mechanically [King 1969, Igarashi et al. 1973, Deutsh 1973]. However,

the programs verified have all been sequential programs and usually small programs. Hoare [Hoare 1969, 1971a] has described an axiomatic approach to proving correctness of sequential programs. In this thesis we have extended Hoare's approach to concurrent programs, in particular programs that are independent except for their interaction through monitors [Hoare 1974]. Further, we have shown for the first time the feasibility of stating and verifying assertions about operating system programs.

The remainder of this thesis is organized as follows:

Chapter 2 discusses the hierarchical levels of abstraction methodology and its application to the many-process problem. It also explains potential problems with non-hierarchical solutions to the many-process problem.

Chapter 3 discusses structured programming and introduces the concept of monitors [Hoare 1974]. It presents the specifications of the four levels of the solution to the many-process problem.

Chapter 4 discusses the techniques developed for verifying concurrent programs and their application to the specifications of the solution to the many-process problem.

Chapter 5 summarizes the results of this thesis and discusses its limitations and possible extensions.

Appendix A contains the axioms for the specification language used, PASCAL [Wirth 1972] extended with monitors.

Appendix B contains the proof of correctness of the simple scheduler, level 1 of the solution to the many-process problem.

Appendix C contains the proof of correctness of the simple memory manager, level 2 of the solution to the many-process problem.

CHAPTER 2. DESIGN

This chapter is organized as follows. In section 2.1, the hierarchical levels of abstraction approach [Dijkstra 1968a, 1968b] to the design of systems is explained. In section 2.2, the design problem to be considered in this thesis is stated. In section 2.3 a two-level solution is given and its demerits discussed. Section 2.4 contains two alternative solutions, each requiring three levels, that are improvements over the two level solution. In section 2.5, a four-level solution is presented that is shown to be more desirable than the three-level solutions under certain conditions. In section 2.6, a non-hierarchical solution is considered and shown to contain a potential deadlock [Coffman et al. 1971] under certain assumptions. The solution and the demonstration of deadlock are presented to indicate the complexity of non-hierarchical solutions that may be necessary to obtain deadlock-free operation. Finally, in section 2.7, the conclusions to be drawn from the design process are summarized.

2.1 HIERARCHICAL LEVELS OF ABSTRACTION

The use of hierarchical levels of abstraction

[Dijkstra 1968a, 1968b] is a method of structuring complex systems, in particular operating systems. In this method the transformation from a given hardware machine A_0 to the desired user machine A_N is conceived as a series of transformations, resulting in a conceptual ordered sequence of abstract machines $A_0, A_1, \ldots A_N$. ("Machine" is used to convey the notion of a hardware machine. The word "abstract machine" conveys the notion that there may not exist real hardware that corresponds to that machine.) The software which transforms machine A_{i-1} to A_i $(0<i\leq N)$ is defined in terms of the machine A_{i-1}, i.e. the functions made available by the machine A_{i-1}, and is said to be the i_th level of the operating system. The software of level i in turn makes available more convenient functions to work with resulting in the abstract machine A_i. Note that a natural restriction in this approach is that the software of level i cannot make use of the functions (or software) of higher levels; further, the resources of the machine A_{i-1} used in transforming it to the machine A_i are no longer available to the higher levels.

The advantage of this approach is that it reduces the problem of designing a collection of programs interacting in an unrestricted manner to the design of a sequence of

subsystems built one on top of the other with well-defined interactions. Parnas [Parnas 1974] has discussed the various uses of the term "hierarchical structure" in the design of operating systems. In his terminology, the THE system possesses a program hierarchy as well as a process hierarchy that happened to coincide. In the system that we will describe, we use the term "hierarchical" in the sense of program hierarchy, that is, a program is at a level higher than or equal to that of another program if it depends on it. A program p_i is said to depend on another program p_j if they operate on common data structures, or if p_i calls p_j and the purpose (effect) of p_j is relevant to the purpose (effect) of p_i. We do not rule out process hierarchies; that is, it is possible for the software of a level to be composed of a set of functions that are used by higher levels as procedures, or a set of processes that are used by higher levels by giving them work through the interprocess communication facility, or a combination of both. It should be noted that neither the programs nor the processes of level i can use the functions or processes of higher levels. Examples of systems implemented using the hierarchical levels of abstraction approach are the THE system [Dijkstra 1968b] and the VENUS system [Liskov 1972].

The methodology is also used in a system being designed at Stanford Research Institute [Neumann et al. 1975]. For example, the levels in the THE system are (lowest to highest):

1. Processor allocation and process synchronization.
2. Memory management (segment controller).
3. Operator console sharing (message interpreter).
4. Input/output management.
5. Independent user programs.

The software of level 1 hides the interrupts; that is, above this level the sharing of a physical processor among a set of sequential processes is invisible and each sequential process can assume that it has a dedicated processor. In the THE system, the higher levels are implemented as sequential processes and use the facilities provided by level 1. To implement sequential processes, the level 1 software needs memory space to remember the process state. In the THE system, only main memory is used by the level 1. For this purpose, and since main memory is generally a scarce resource, the number of sequential processes among which level 1 shares physical processors is limited to a small number, (10 - 15). The software of level 2 implements "virtual memory". Processes (programs)

above this level are written without concern as to whether their address space is in the main memory or in the secondary memory. The memory manager thus provides a convenient large address space. The software at level 1 is restricted to using only main memory because the basic machine did not conveniently provide a large address space for its use, and it could not use the facilities of the segment controller because of the hierarchical restriction.

2.2 THE MANY-PROCESS PROBLEM

The many-process problem is to design an operating system with the following characteristics: (1) the number of concurrent processes is large, (2) each process may have a large address space.

By a large number of processes, we mean a number so large that it is undesirable (given the current state of technology) to keep the necessary state information of all processes in the main memory.

By a large address space, we mean an address space larger than available main memory space, so that it is necessary to use secondary memory space. We will call the large address space, "virtual address space" to denote that it may not be completely resident in the main memory. We will not consider the design of a complete operating system but

only of those parts that deal with processor and main memory sharing. These parts are common to and are usually similar in most operating systems.

One motivation for choosing this problem is that at present there are few systems that support a very large number of processes. Examples of systems which support a large number of processes are OS-MVT [Katzan 1973] and MULTICS [Organick 1972]. These systems, although allowing dynamic creation of processes, keep the process states in the main memory at all times and the number of concurrent processes allowed by the system is a function of the system load. We feel that the ability to create a large number of processes is useful in structuring solutions to problems which lend themselves naturally to be structured as many independent processes, for example, transaction systems and weather forecasting.

Another motivation was to consider a problem with a constraint that violates the hierarchical organization of systems like the THE and VENUS systems, with the aim of investigating the use of the hierarchical levels of abstraction methodology.

2.3 SOLUTION 1

From the statement of the problem it is clear that a

solution may be divided into two major parts: (1) the sharing of physical processors among concurrent processes, and (2) the creation of virtual address spaces. The software implementing the sharing of physical processors among processes and facilities for process synchronization (process synchronization is explained later) is called the scheduler. The software implementing virtual address spaces is called the memory manager.

Scheduler: The scheduler is responsible for sharing physical processors among a set of concurrent processes; further, the fact that a process is sharing a processor with other processes is invisible to the process itself.

In sharing a processor among processes, the scheduler has to save the "state" of the process at the time the processor is being taken away. The state is the information in the temporary storage of the processor, such as registers, that is relevant to the execution of the process. This information has to be saved because when the process resumes execution on the processor, the temporary storage of the processor should contain the same information as when the processor was preempted. This information, if not saved, would be lost because the temporary storage is loaded with the state corresponding to the

process that would be executing on it next. Thus the scheduler needs some memory space to save a process state, and the amount of memory space needed is directly proportional to the number of processes among which the physical processors are shared.

The scheduler also provides facilities for process synchronization. Two (or more) concurrent processes, i.e. processes executing logically in parallel and with undetermined speed ratios, may desire to synchronize their activities. One of them may wish to wait for an event until the other process signals that event. When a process wishes to wait for an event, it is desirable to preempt the processor and allocate it to another process; a similar action may be needed when a process signals an event. Since the actions required in providing synchronization are closely related to those required for processor allocation, we have included them as part of the scheduler.

Memory Manager: The memory manager is responsible for creating the virtual address space for a process, that is, it creates the illusion for the process that its address space is wholly resident in executable memory (main memory), when in fact part or all of it may be in nonexecutable memory (secondary memory). The memory manager

creates the illusion of virtual address space by mapping a virtual address onto a real (main memory or secondary memory address) address and transferring information between the main memory and secondary memory as required. When a process needs information in a virtual address that is currently absent from the main memory, it has to wait until the information is brought into the main memory. Since the process is designed not to consider the possibility of information being absent from the main memory (at least logically, although it may use this information for its efficient organization), it cannot be allowed to proceed until the information needed is available in the main memory.

Since the scheduler needs a large address space and the memory manager is designed to provide large address spaces, it is natural to consider the following ordering of the levels as a solution to the many-process problem:

level 1: Memory manager

level 2: Scheduler

The disadvantage of this solution is the following. Given the current state of computer hardware technology, the time to transfer information between the secondary memory and the main memory is very large (approx. 25 msec.)

compared to the time to switch processes on a processor (25 micro sec.) and the time to execute an instruction on a processor (1 micro sec.), so that with the ordering as in level 1, the processor will be idle whenever there is need to transfer information between the main memory and the secondary memory. This is so, because the memory manager is at a lower level than the scheduler and cannot, by the hierarchical restriction, use the facilities of the scheduler to cause the processor to be switched to another process.

The conventional ordering of levels, i.e. the scheduler at level 1 and the memory manager at level 2, will not work because the scheduler needs a large address space, and if it is not to use the facilities of the memory manager it will have to provide its own.

Hence, although solution 1 satisfies our design goals, which did not include any efficiency considerations, we observe that even considering the probable performance of the system from a broad qualitative aspect it will be undesirable. Thus, we add the following constraint to be met by our solutions and look for other solutions.

The nonbusy waiting constraint: A processor should not be kept unduly idle waiting for the completion of

information transfer between the main memory and secondary memory. We emphasize that the constraint uses the term "unduly idle" rather that just "idle" because it may be advantageous in certain cases to keep the processor idle for some time (e.g. when the idle time is less than process switching time) and we wish to rule out grossly inefficient solutions, based on qualitative reasoning.

It is perhaps more desirable to quantify the term "efficient", but a quantitative evaluation of efficiency would require an analysis of system operation and our aim is to prune out solutions that are likely to be undesirable before investing time and effort in actually constructing the system.

In the next two sections, we consider alternative solutions which, on the basis of qualitative reasoning and the assumption that it is a desirable goal to satisfy the nonbusy waiting constraint, are shown to be better than solution 1.

2.4 THREE-LEVEL SOLUTIONS.

As discussed in the previous sections, neither of the two possible two-level solutions satisfy our requirements and there seems to be a need for a loop in the structure of our solution, that is, the scheduler and the memory

manager each seem to require the facilities provided by the other. We can cut this loop in two ways, either by providing two levels of software each providing the functions of the scheduler, or by two levels each providing the functions of the memory manager. We will discuss both of these three-level solutions.

Solution 2a. Consider the following solution:

level 1: Simple scheduler

level 2: Memory manager

level 3: Scheduler

The simple scheduler provides the same facilities as the scheduler in solution 1, except that it does so for a small number of processes -- small enough that all the process states can be kept in the main memory at all times; thus the simple scheduler does not need the facilities provided by the memory manager.

The memory manager provides the same facilities as the memory manager in solution 1.

The scheduler also provides the same facilities as the simple scheduler except it does so for a large number of processes and uses the facilities provided by the memory manager to obtain a virtual address space. Further, instead of sharing the physical processor among a large number of

processes, it is conceptually better to view the scheduler as sharing virtual processors, created by the simple scheduler, among the processes. Thus the scheduler also uses the facilities of the simple scheduler. The advantage of conceiving the scheduler as sharing virtual processors rather than physical processors obtained from the simple scheduler is that the facilities of the simple scheduler need not be used by the processes at levels higher than the scheduler, and we can avoid potential deadlocks that can arise if the scheduler and the processes at higher levels that use the scheduler facilities also use the simple scheduler facilities. Such deadlock situations may arise because the scheduler may itself be designed as a sequential process and may be blocked in attempting to preempt a higher level process which in turn could be blocked waiting for some facilities of the scheduler.

The disadvantages of the solution 2a are the following:

1) The number of processes that can concurrently share the physical processors is limited to the number of processes that are managed by the simple scheduler; e.g. if the simple scheduler provides facilities for ten processes and if all ten processes are waiting for information to be brought from secondary memory into the

main memory, then the physical processors will be idle.

2) The facilities provided by the memory manager are used by the scheduler as well as higher level processes. Although they both need the same logical interface, their performance requirements may be quite different and might require different data structures and algorithms to be used by the memory manager.

Solution 2b: Now consider the other three-level solution:

level 1: Simple memory manager

level 2: Scheduler

level 3: Memory manager.

In this solution the scheduler and the memory manager are the same as in solution 1. The simple memory manager provides the same facilities as the memory manager, except that it is used solely by the scheduler.

This solution does not have the disadvantages of solutions 1 or 2a. The disadvantage of this solution is that when the scheduler needs information which is not currently in the main memory, the physical processor is kept idle while the information is being transferred from the secondary memory. In the next section we will con-

consider a four-level solution that removes this deficiency.

2.5 FOUR-LEVEL SOLUTION

We propose the following solution to the many process problem, that does not have the disadvantages of solutions 1, 2a, 2b.

level 1: Simple Scheduler

level 2: Simple memory manager

level 3: Scheduler

level 4: Memory manager

The simple scheduler and the scheduler are the same as in solution 2a and the simple memory manager and memory manager the same as in solution 2b. This solution does not have the disadvantages of solution 2a because there are two separate memory managers. The simple memory manager is used solely by the scheduler. These memory managers do not share any main memory or secondary memory. Thus this solution does not have the second disadvantage of solution 2a. The memory manager uses only the facilities of the scheduler; as a result, when higher level processes are waiting for information transfer, the scheduler which has a large address space can switch the virtual processor to another higher level process (of which there are many). Thus this

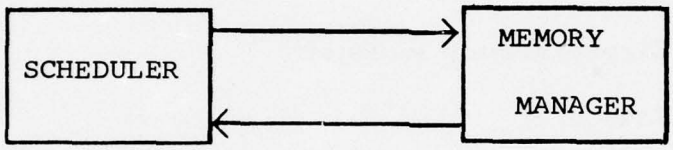


Figure 2.1

A Non-hierarchical Solution

solution does not have the first disadvantage of solution 2a. In this solution, the physical processors will be idle only if all the processes that constitute the scheduler level are waiting for information transfer to be performed by the simple memory manager. The probability of a scheduler process waiting for information transfer is expected to be much less than that of a higher level process waiting for information transfer, and by choosing an appropriate number of processes to construct the scheduler level, we can reduce the physical processor idle time to much below that expected in solution 2b (at an increase in overhead).

We will consider the four level solution in more detail in the next chapter and also give a specification of each of the four levels.

2.6 A NON-HIERARCHICAL SOLUTION (WITH POTENTIAL DEADLOCK)

In this section we will show that an attempt at obtaining a non-hierarchical solution that satisfies the nonbusy waiting constraint and is modular involves a potential deadlock. Consider the solution shown in Figure 2.1. The scheduler and the memory manager provide the same functions as in solution 1 but are permitted to use each other's facilities. If we hope to be able to

understand the resulting system, it is important to keep the interactions between the modules as simple as possible. Hence we make the following assumptions regarding the design of the two modules.

Assumptions:

A1: The scheduler and the memory manager are independent modules.

A2: The memory manager handles requests from the scheduler identically to requests from other users.

A3: The scheduler handles requests from the memory manager identically to requests from the other users.

The assumption A1 is made to simplify the interactions between the two modules. By independent modules, we mean that in the specification and implementation of one module no use is made of the knowledge of implementation of the other module. This is in keeping with the criteria for decomposing systems into modules suggested by Parnas [Parnas 1972].

The assumptions A2 and A3 are made so that there is in fact a single memory manager module and a single scheduler module and not two separate memory managers (as

in solution 2a), or two separate schedulers (as in solution 2b) or two separate schedulers disguised as one. If there is special handling in one module for requests from the other module, then it is better to recognize the inherent structure which is really hierarchical rather than the assumed non-hierarchical structure.

Given the above assumptions we will demonstrate that any solution which satisfies them contains a possible deadlock. Consider the following sequence of events:

1. A higher level process (process A) other than scheduler or memory manager requires information currently absent from the main memory. The memory manager initiates the transfer of information and passes control to the scheduler to schedule another process on the processor.
2. The scheduler in attempting to schedule another process needs information currently absent from the main memory. Note that this is possible because the scheduler is assured a large address space by the memory manager and uses it just like any other higher level process. (It follows from A1-A3 that until the information needed is brought into the main memory, no other process can be scheduled. Note the processor can enter into an idle

loop, waiting for an interrupt.)

3. The information needed by the process A has been transferred into the main memory and the memory manager tries to call the scheduler function responsible for resuming process A. (This is an example of the use of process synchronization functions of the scheduler. The memory manager can gain control of the processor by virtue of an interrupt.)

4. The system is in a deadlock, because the scheduler is unable to accept the request from the memory manager to resume process A until the information needed by the scheduler is brought into the main memory, but the memory manager may be unable to bring the information needed by the scheduler until the scheduler marks process A as eligible for being resumed.

Note that it may be possible to obtain a non-hierarchical solution by combining the two modules into one and violating assumption A1, but it might be difficult to understand such a solution and to demonstrate the absence of a deadlock might be even more difficult.

2.7. CONCLUSIONS:

The conclusions to be drawn from the discussion in

this chapter are the following. It is possible to obtain hierarchical solutions to complex problems, like the many-process problem, although a straightforward application of the levels of abstraction approach may lead to unacceptably inefficient solutions. It has been shown that by a systematic application of the methodology efficient hierarchical solutions can be devised. It has also been shown that even at a very early stage of design, there are some appropriate issues of efficiency that need to be considered (like the nonbusy waiting constraint). It has also been shown that a modular but non-hierarchical approach to design may lead to solutions with potential problems, such as the deadlock in the non-hierarchical solution to the many-process problem.

CHAPTER 3. SPECIFICATION

The purpose of this chapter is to present a structured specification of the four-level solution given in section 2.5. This chapter is organized as follows. In section 3.1, a brief introduction to the concepts of structured programming is given. In section 3.2, the concept of a monitor is explained. Sections 3.3 to 3.6 contain the specification of the four levels of solution 3 given in section 2.5. The programming language PASCAL extended with monitors [Hoare 1974] is the language used for the specification. The reasons for choosing PASCAL as the specification language are the following: a) PASCAL was designed to aid in structured programming; b) PASCAL has been axiomatized [Hoare and Wirth 1972] and since one of the goals of this thesis is to verify the correctness of the specifications, the choice of PASCAL will simplify the task of verification; and c) PASCAL is a machine independent language and so the specification can also be machine independent. In section 3.7, the conclusions to be drawn from the exercise of obtaining a structured specification of an operating system are summarized.

3.1. STRUCTURED PROGRAMMING

It is difficult to give a precise definition of structured programming. The concepts of structured programming have been explained by means of small examples in the book Structured Programming [Dahl, Dijkstra, and Hoare 1972]. A good explanation of the nature of structured programming is given in an article by Knuth [Knuth 1974]. A definition of structured programming, attributed to Hoare by Knuth [Knuth 1974], is: "the systematic use of abstractions to control a mass of detail, and also a means of documentation which aids program design."

In our view, structured programming is a methodology for structuring the data and the flow of control of the programs, so that the resulting program is "easy" to understand and verify. An important concept in structured programming is that of abstraction, both data abstraction and procedure abstraction. The use of abstractions enables us to decompose large systems into smaller constituent systems and their interactions. We do not need to understand how the properties of the constituent systems are themselves obtained. Thus, use of abstractions helps to cut the mass of deatil down to manageable proportions.

The advantages of structured programming can be

obtained only by discovering abstractions that help reduce the total amount of information (state information) needed to understand a system. A guide to obtaining appropriate abstractions is to use the concept of data abstractions. In programs, we generally use primitive data types to construct data structures that are used for their abstract properties; for example, we may use an array of integers as a queue. To use the concept of data abstractions we should consider the abstract properties of queues and construct procedures and functions to operate on the array of integers to provide the abstract properties. The array of integers should not be used for other purposes and the array should not be accessed by any program other than the procedures and functions used to implement the abstract data type queue. If these restrictions are followed, the data can be generally expressed as an invariant relation and its purpose becomes clearer.

The use of procedure abstraction also helps in making programs more understandable because, once the working of the procedure is understood in terms of input/output relations on its parameters, we may use the procedure at many places and just use the input/output relations to understand its effect. This technique is useful if we

construct procedures with input/output relations which can be used at many places in the program.

Another concept in structured programming is the use of appropriate and simple control structures. It has been pointed out in the literature [Dijkstra 1968c, Knuth 1974, Wirth 1974] that go-to statements make it difficult to understand the flow of control and the appropriate use of control structures like if-then-else and while statements make its use unnecessary. For an explanation of the use of the control structures refer to [Wirth 1974].

3.2. THE MONITOR CONCEPT

A monitor may be viewed as an abstract data structure that allows concurrent processes exclusive access to shared variables. A monitor may also be viewed as a resource allocator using the shared variables to administer the resource allocation policy. Using the notation of PASCAL, a monitor may be declared as shown in Figure 3.1. It consists of data variables local to the monitor and procedures and functions, which operate on these variables. These variables may be accessed only by the procedures and functions declared to be in the monitor. Although the monitor is shared among concurrent processes, the execution of the monitor procedures and functions exclude each other

```
monitor monitorname;
  begin...declarations of data local to the monitor;
    procedure procname (...formal parameters...);
      begin...procedure body...end;
    ...declarations of other procedures local to the
          monitor;
    ...initialization of local data of the monitor...
  end;
```

Note that the procedure bodies may have local data, in the normal way. In order to call a procedure of a monitor, it is necessary to give the name of the monitor as well as the name of the desired procedure, separating them by a period:

```
monitorname.procname(...actual parameters...);
```

Figure 3.1

Monitor declaration

in time.

To provide synchronization facilities, monitors may contain a new type of variable, known as the type condition. A condition variable has no stored value accessible to the program. A condition variable is represented by an (initially empty) queue of processes waiting on the condition. The only operations allowed on a condition variable are "wait" and "signal". The syntax for these operations is cv.wait and cv.signal, where cv is a condition variable. The effect of cv.wait is to suspend the execution of the process invoking the wait operation and to release access to the monitor. The process will be resumed on a signal operation on the same condition variable by another process. The effect of cv.signal is as follows: if the queue of processes waiting on the condition cv is empty, then there is no effect; if the queue of processes waiting on the condition cv is nonempty, then one of the waiting processes is removed from the queue and its execution resumed. The process invoking the signal operation is suspended. It will be resumed when exclusion on the monitor is released by the process which has been resumed or by some other process. The process invoking signal is added to the queue of processes waiting to gain access to the monitor. If the

signal is the last operation to be executed before exiting from the monitor, then these two operations may be combined into a single operation.

The monitor concept has also been described by Brinch Hansen [Brinch Hansen 1973, 1975]. Examples of the use of monitors and condition variables are given by Hoare [Hoare 1974]. The class notation of SIMULA 67 [Dahl and Hoare 1972] may also be used to declare several monitors with identical structure and behaviour but using different variables.

3.3. SIMPLE SCHEDULER

In this section the specification of the simple scheduler is given. We will first give an explanation of the intended use of the simple scheduler and then the specifications.

The simple scheduler is completely resident in the main memory. It implements processor allocation and process synchronization for a small and fixed number of processes whose state information is always in the main memory. The processes at this level are the processes required to implement the simple memory manager and the scheduler. These processes are permanent and it is assumed that they are initialized properly. The simple scheduler is implemented as re-entrant procedures shared by all the processes

it schedules. In order for the simple scheduler to perform its function it must have mutually exclusive access to its own data. Thus it is necessary to implement a mutual exclusion mechanism [Dijkstra 1968b, Brinch Hansen 1973]. In a single processor system with interrupt-controlled input/output, it is sufficient to be able to turn off and turn on interrupts in order to achieve mutual exclusion. In a multi-processor environment, mutual exclusion is usually achieved through provision of a special instruction such as the test and set instruction in the IBM 360 machines [IBM]. Mutual exclusion is implemented by the hardware at the lowest level. Mutual exclusion in the simple scheduler is achieved by entering a special state that is called the mutual exclusion state.

A processor attempting to enter the mutual exclusion state while it is occupied by another processor will be delayed until the earlier processor exits from the mutual exclusion state. The delayed processor simply busy waits, that is, it tries again and again to enter the mutual exclusion state until the hardware enables it to do so. The primitives at the simple scheduler level must be so implemented that the time spent in the mutual exclusion state is as little as possible. This is more important

for multi-processor environments than for uni-processor environments, because in multiprocessor environments physical resources (processors) are left idle due to busy waiting.

To avoid confusion, the term "enter_mutual_exclusion_state" is used to specify that the mutual exclusion state is entered, with the understanding that if any other processor is in the mutual exclusion state, the processor will busy wait and all interrupts are disabled for that processor. To leave the mutual exclusion state, the term "exit_mutual_exclusion_state" is used. All instructions between "enter_mutual_exclusion_state" and "exit_mutual_exclusion_state" constitute a single primitive; that is, they are provided mutual exclusion.

As suggested by Hoare [Hoare 1973], monitors are used to achieve mutual exclusion and condition variables are used for process synchronization. Mutual exclusion can be implemented for monitors by the use of boolean semaphores initialized to the value true. Associated with each monitor is a boolean semaphore. The operation enter (monitorname) corresponds to the operation P(monitorname) on the associated semaphore, monitorname. The operation exit(minotorname) corresponds to the operation

V(monitorname) [Hoare 1974].

A brief description of the primitive operations is given below:

Let m be a monitorname. The effect of enter(m) is to test the value of m. If m has the value true, then m is set to false and the process executing enter(m) is allowed to proceed further. If m has the value false, then the process is suspended and put on a waiting list associated with m. The process is resumed when another process executes an exit(m) and is then allowed to proceed to the instruction after enter(m). The effect of exit(m) is to check the waiting list associated with m and, if any process is on that list, to select one of them and allow it to continue. If no process is on the waiting list associated with m, the value of m is changed to true. In either case the process executing exit(m) is allowed to continue.

Condition variables serve a function similar to that served by Brinch Hansen's "await" primitive [Brinch Hansen 1973]. The operations on condition variables were explained in section 3.2.

One of the purposes of the simple scheduler is to share a set of processors among a set of concurrent

processes. It is assumed that the number of concurrent processes is larger than the number of processors. If this assumption is not valid, the solution is not invalidated, but a simpler solution can be obtained (with fixed processor allocation). It is assumed that the number of concurrent processes is N, and that these processes are identified by the integers 1 to N. A process can be in one of three states: active, executing on a processor; ready, waiting for a processor; blocked, waiting on a monitor or condition variable. The fact that a process is ready or running is invisible to the process itself, since it is unaware of the distinction. When a processor becomes free, one of the ready processes is selected to run on that processor. The criteria by which the process is selected is known as the scheduling policy. In our specification we have not specified this scheduling policy; instead we call upon the procedure dispatch which implements the desired scheduling policy. The scheduling policy is not specified because it may vary widely among different systems. The procedure dispatch is also responsible for saving the state of the process freeing the processor and loading the processor with the state of the process that will run next on the processor. Since we always call upon the

procedure dispatch, whenever the processor is released by the currently executing process (or a scheduling decision is to be made), we assume that there exists at least one available ready process. Hence, there is an idle process that gets dispatched only when no other process is ready and puts the processor in an idle loop, waiting for an event to occur that will make one of the blocked processes ready. For multiprocessor configurations there would be as many idle processes as there are physical processors.

The specification of the procedure dispatch is not given because it is dependent on the scheduling policy. The purpose of the procedure dispatch is to share processors among the processes. Since this sharing is to be invisible to processes above this level, the effect of the procedure dispatch is not relevant to the input/output assertions to be proved of this level for the use of higher levels. However, the effect of dispatch is relevant to certain properties of this level, such as the property that the processes do get processor time, which is an implicit assumption made at higher levels.

The specification of the other procedures is given in section 3.3.1. In section 3.1, we mentioned the use of structured programming in specifying the programs. We

would like to emphasize the use of a data abstraction in the specification of this level that helped in simplifying the task of verifying the specifications. We have used an abstract data type called distinct element queue, which is a queue such that all elements of the queue are distinct. This is the type of queues we need to associate with monitors and condition variables. The specifications of the procedures for monitor entry and exit and the procedures for operations on condition variables make use of this abstract data type and the abstract operations on it. In our verification we verified the properties of this abstract data type only once and used the verified properties in all the procedures. Thus the use of abstract data types not only made the task of specifying the procedures simpler but also eliminated the need for verifying similar programs repeatedly.

Interrupts

In most computer systems there is an asynchronous signaling of conditions that is implemented in hardware. These conditions are called interrupts. One can view the interrupts as semaphore variables implemented at a lower level, that is, the hardware. The interrupts are used mainly by devices connected to the central processing unit

by hardware, and the detection of such signals is also done by hardware. Since the actions taken by the hardware on the detection of an interrupt (transfer of control to a particular location in the memory) require control of the processor, which is not available to higher (above simple scheduler) level processes, it is advisable to mask the interrupt mechanism and to convert interrupts to operations on semaphore variables. We will use logical variables to indicate if an interrupt had occurred since the last time it was accepted. Thus interrupts may be considered as variables of type monitorname defined in the monitor mutual_exclusion_state with the in_use field indicating if the interrupt is pending.

At level 0, the hardware is assumed to check for interrupts (including traps) after every instruction execution cycle (in some processors, after every memory access), and if an interrupt is present, the processor fetches the next instruction from a pre-assigned location. The effect of the interrupt is to insert a call on the procedure signal_interrupt in the instruction stream of the currently running process. The procedure signal_interrupt converts the interrupt into an exit operation (V on a semaphore) on the monitorname variable associated with the interrupt. A process desiring to wait on an interrupt invokes the

procedure wait interrupt. The procedure wait_interrupt has the same effect as the enter operation on the (P on a semaphore) monitorname variable associated with the interrupt. There is no difference between the procedures signal_interrupt and exit, and the procedures wait_interrupt and enter.

3.3.1. Specification of Simple Scheduler Programs.

The data structures and procedures for the implementation of monitors and condition variables are now specified. The integer variable p denotes the active process that has invoked the scheduler procedures. Note that a process can wait on at most one monitor or condition variable. The syntax for calling a monitor procedure is monitorname.procedurename. This is translated as enter (monitorname,p); procedurename; exit(monitorname,p);. The syntax for a wait operation on condition variable cv is cv.wait. This is translated as a call on procedure wait with the parameters cv, m, and p. The variable m is the monitor name within which condition variable cv is declared. Similarly, the operation cv.signal is translated as a call on the procedure signal with parameters cv, m, and p.

The structure of the simple scheduler is given in Figure 3.3.

```
{declarations for the simple scheduler}

type processid = 1..N; {N, an integer, is the number of
                        processes at this level ;}
     monitorname = record in_use: Boolean;queue: 0..N end;
     condition = 0..N; {0 is used as the null value for
                        queue and condition};

var  plink: array [processid] of 0..N; {processid, null};
     status: array [processid] of (active, ready, blocked);

function first(t: 0..n): 1..N;
begin
   if t ≠ 0 then first := t else ERROR;
end first;

function is_empty(t: 0..N): Boolean;
begin
   is_empty := (t = 0);
end is_empty;

procedure remove(var t: 0..N);
begin
   if t ≠ 0 then t := plink[t] else ERROR;
end remove;

procedure append(var t: 0..N; d: 1..N);
var x: 1..N;
begin
   plink[d] := 0;
   if t = 0 then t := d
   else
      begin
         x := t;
         while plink[x] ≠ 0 do x := plink[x];
         plink[x] := d;
      end;
end append;
```

Figure 3.2

Simple Scheduler Specification

```
procedure enter (var m: monitorname; p: processid);
begin
   enter_mutual_exclusion_state
   if ¬(m.in_use) then
   m.in_use := true
   else
      begin
         append(m.queue);
         status[p] := blocked;
         dispatch(p);
      end;
   exit_mutual_exclusion_state
end enter;


procedure exit(var m: monitorname; p: processid);
begin
   enter_mutual_exclusion_state
   if is_empty(m.queue) then
   m.in_use := false
   else
      begin
         status [first(m.queue)] := ready;
         remove(m.queue);
         status[p] := ready;
         dispatch(p);
      end;
   exit_mutual_exclusion_state
end exit;
```

Figure 3.2 (continued)

Simple Scheduler Specification

```
procedure wait (var m: monitorname; var cv: condition;
                p: processid);

begin
   enter_mutual_exclusion_state;
   append (cv,p);
   status[p] := blocked;
   if is_empty(m.queue) then
      m.in_use := false
   else
      begin
         status[first(m.queue)] := ready;
         remove (m.queue);
      end;
   dispatch (p);
   exit_mutual_exclusion_state;
end wait;
```

Figure 3.2 (continued)

Simple Scheduler Specification

```
procedure signal (var m: monitorname; var cv: condition;
                  p: processid);

begin
   enter_mutual_exclusion_state
   if ¬(is_empty(cv)) then
      begin
         append(m.queue, p);
         status[p] := blocked;
         status[first(cv)] := ready;
         remove(cv);
         dispatch(p);
      end;
   exit_mutual_exclusion_state
end signal;
```

Figure 3.2 (continued)

Simple Scheduler Specification

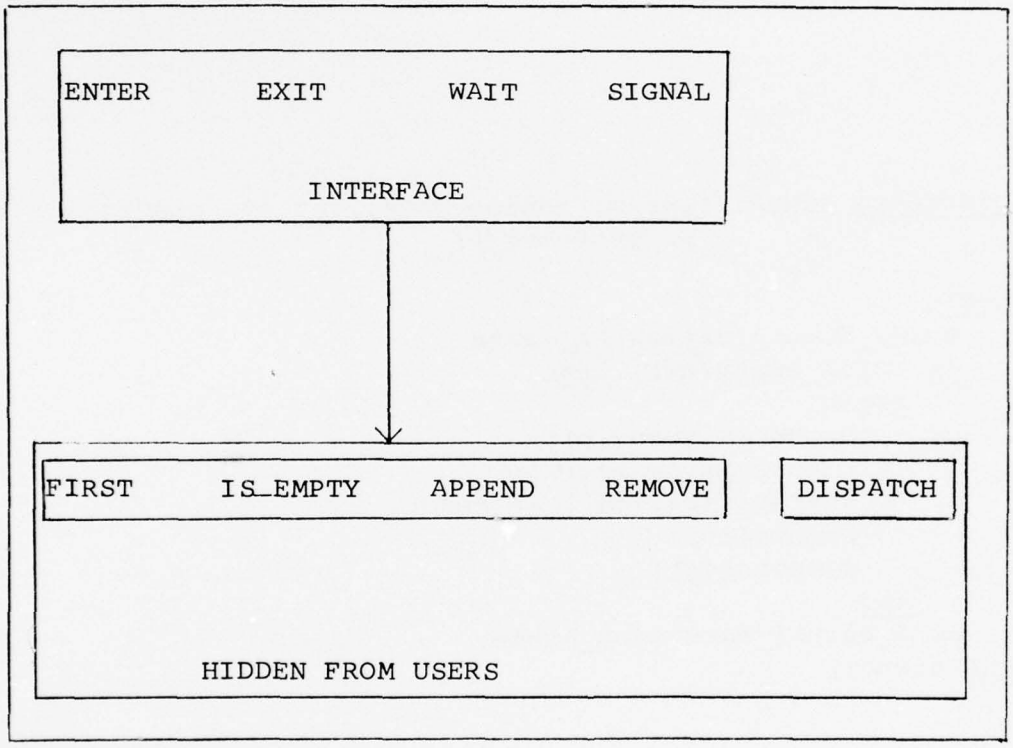


Figure 3.3

Structure of the Simple Scheduler

3.4. SIMPLE MEMORY MANAGER

The purpose of the simple memory manager is to create the abstract object virtual memory from the real objects main memory and secondary memory. The virtual memory is to be created by sharing the main memory and secondary memory among a number of users in such a way that the users need not be aware as to the real location of the information in the main memory. The operations defined on virtual memory are clear_page, fetch and assign. These operations have to be implemented in terms of operations on real memory. It is a basic assumption that the contents of virtual memory may be read or altered by a user only when the information is present in the main memory. It is also assumed that all the information in the virtual memory may not be residing in the main memory. Since the virtual memory as well as the main memory and secondary memory are shared among users, we will use the monitor concept to control access to the shared resources. The use of monitors implies the use of the monitor primitives implemented at level 1.

In the following sections, we present a specification of the simple memory manager, starting from the most abstract version in section 3.4.1 to the final version in

section 3.4.9. In each section we transform the specification to improve its realization on currently available hardware. Section 3.4.10 concludes the discussion of the simple memory manager.

3.4.1. The abstract object virtual_memory

The abstract object virtual_memory is considered to be an array of virtual_page_frames. Each virtual_page_frame, itself being an array of lines. Each line contains a word. Thus the virtual memory is defined to consist of V page frames of L lines each. Let

```
line = 0..L-1; {L = number of lines in a page frame.}
vpf = 0..V-1; {V = number of virtual memory page
                   frames.}
mmpf = 0..M-1; {M = number of main memory page frames.}
smpf = 0..S-1; {S = number of secondary memory page
                   frames.}
virtual_memory_address = record vpno: vpf; offset:
                          line end;
main_memory_address = record mmpno: mmpf; offset:
                          line end;
secondary_memory_address = record smpno: smpf;
                            offset: line end;
virtual_memory: array[virtual_address] of word;
main_memory: array[main_memory_address] of word
secondary_memory: array[secondary_memory_address] of
                   word;
```

We use the abbreviation mm for main_memory and sm for secondary memory respectively. By

mm[mma.mmpno], where mma: main_memory_address we denote the contents of the main memory page frame mma.mmpno.

Similarly by

sm[sma.smpno] we denote the contents of the secondary memory page frame sma.smpno

thus vm[v] = sm[s] = mm[m] means that the contents of the virtual page frame v are the same as the contents of the secondary memory page frame s and the main memory page frame m.

The operations clear_page, fetch and assign are defined using Hoare's notation, and vmo and wo as free variables as follows. We use the abbreviation vm for virtual_memory.

vm=vmo{clear_page(v:vpf)} vm=(vmo,v:0)

vm=vmo {fetch(va:virtual_address;var w:word)}
w=vm[va] ∧ vm=vmo

vm=vmo {assign(va:virtual_address; w:word)}
vm=(vmo,va:w) ∧ w=wo

The data structures and the procedures that implement the abstract object virtual_memory are given in figure 3.4.

3.4.2. Virtual_memory_version 1: Introduction of page replacement

The statement

out: w=sm[sm_adr, va.offset]

```
type vp_state = record loc: (in, out, all_zero);
                mm_adr: 0..M-1; sm_adr:0..S-1 end;

var virtual_map: array [0..V-1] of vp_state;

with the invariants:

virtual_map[v].loc = in iff mm[mm_adr] = vm[v] ^
virtual_map[v].loc = out iff sm[sm_adr] = vm[v] ^
virtual_map[v].loc = all_zero iff vm[v] = 0

procedure clear_page (v: vpf);
begin
   virtual_map[v].loc := all_zero;
end clear_page;

procedure fetch(va: virtual_address; var w: word);
begin
   with virtual_map[va.vpno] do
      case loc of
         in: w := mm[mm_adr, va.offset];
        out: w := sm[sm_adr, va.offset];
        all_zero: w := 0;
   end;
end;

procedure assign(va: virtual_address; w: word);
begin
   with virtual_map[va.vpno] do
      case loc of
         in: mm[mm_adr, va.offset] := w;
        out: sm[sm_adr, va.offset] := w;
        all_zero: va[va] := w;
      end;
end;
```

Figure 3.4

Virtual_memory Version 0

in the procedures fetch and assign is not directly executable in the present computer systems. The information has to be brought into the main memory before it can be accessed by the central processing unit.

The statement may be implemented in terms of other executable statements as follows.

```
out: begin  Find a free main memory page_frame;
            copy information from the smpf sm_adr
            to the main memory page frame;
            update vp_state to satisfy the invariant;
            access w in the main memory page_frame;
     end;
```

That is, by

```
out: begin  mm_acquire (m);,
            input (m,sm_adr);
            loc := in;
            w := mm[m,va.offset];
     end;
```

where mm_acquire is a procedure that returns the resource free mmpf; i.e. a mmpf which does not contain any valid virtual_memory contents. Input (m,sm_adr) is a procedure that copies the contents of the secondary memory page frame sm_adr into the main memory page frame m.

If we have to find a free main memory page frame, then, from the fact that the number of main memory page frames is much smaller than the number of virtual pages, it follows that we will have to release a main memory page

frame currently containing the contents of some virtual page. The policy to be used in deciding which page frame to release (by first copying its contents onto a free secondary memory page frame) is known as the replacement policy. The choice of replacement policies and the associated problem of models for program behavior is a wide area of research in itself [Coffman and Denning 1973]. For our purposes we have described the memory management system using a cyclic discarding algorithm. It is a fair algorithm and requires very little overhead to implement. We emphasize that it is easy to modify the replacement policy because of the modular nature of our specification. In fact, the replacement policy is specified as an autonomous process.

Initially we will consider a simple replacement policy. We will release a main memory page frame every so often, and do so independently of the demand for free page frames. For this purpose, we will design a process called automatic discarder that will release page frames in order, using the procedure throwout (described later). When a page frame is released it is added to the pool of free main memory page frames, by the procedure mm_release, and can be later allocated by the procedure mm_acquire.

The procedure throwout may be described as follows:

```
procedure throwout (m: mmpf);
begin
   find the virtual page, if any, whose contents are
         currently in the mmpf m;
   if there is no such page then do nothing
   else copy the contents onto a free smpf;
   release the mmpf;
end;
```

We find that in order to find if a virtual page is currently occupying a mmpf and its identity, we have to search through the entire array. It is easier if we define another array m_adr that keeps track of this information.

```
m_adr: array[mmpf] of vpf;
```

The case when the page is all_zero needs to be modified, because, when the contents are all_zero, there is no representation of the virtual page in main memory or secondary memory. Thus before assigning a value to the virtual address, we have to allocate a free mmpf to the virtual page corresponding to the virtual address and initialize it to all_zero. We will also allocate a free smpf to the virtual page so that the case of this page is no different from other non-zero virtual pages, and we will use the allocated smpf to store the virtual page on secondary memory if we have to release the mmpf it

currently occupies.

We will use the procedure sm_acquire to obtain a free smpf and the procedure sm_release to release a free smpf. These procedures are similar to the procedures mm_acquire and mm_release respectively.

We will also modify the procedure fetch so that if the loc = all_zero, the actions performed are similar to those in case of procedure assign; that is, we will allocate a free mmpf initialized to all_zero and a free smpf, so that all memory accesses occur only when loc = in, that is, when the virtual page contents are in the main memory.

We have not considered so far the case of a virtual page that is no longer needed. Such virtual pages are represented in our system with their contents changed to all_zero. (We could have chosen some other value instead of zero; we need some value to express the undefined state.) For this purpose we will use the procedure clear_page defined earlier. The procedure clear_page is part of the monitor virtual_memory and is accessible to processes the same way as the procedures fetch and assign.

The use of condition variables in the procedures mm_acquire, mm_release, sm_acquire, and sm_release, introduces the possibility of multiple faults on the same

virtual page as follows. When a user process is waiting for a virtual page to be brought into main memory from secondary memory, another user process can attempt to access the same page. We cannot block access of all processes to the monitor, because the process automatic_discarder needs to access the procedure throwout, so that it can release a main memory page frame. Note that the process automatic_discarder does not use any other procedures. We can prevent all processes from gaining access to the monitor by the introduction of a condition variable monitorbusy, and a boolean variable busy that is set to true when a user process is in the monitor and to false when no user process is in the monitor. Note that the procedure throwout never attempts to alter the status of a virtual page that is not currently in the main memory and need not wait on the monitorbusy condition.

We will now rewrite the procedures of the monitor virtual_memory to reflect the above mentioned modifications. The monitor is specified in figure 3.5.

3.4.3. Virtual_memory version 2: Introduction of parallelism

We have so far not specified the procedure input and the procedure output, although we have described their

```
monitor virtual_memory;
begin
   type vp_state = record loc(in, out, all_zero);
                    mm_adr: 0..M-1; sm_addr: 0..S-1 end;
   var virtual_map: array [0..V-1] of vp_state;
      m_adr: array [0..M-1] of 0..V;
      m_pool: set of mmpf;
      s_pool: set of smpf;
      busy: Boolean;
      m_nonempty, s_nonempty, monitorbusy: condition;

{Initially }
   m_pool := [0..M-1];
   s_pool := [0..S-1];
   busy := false;
   for v := 0 to V-1 do
      virtual_map[v].loc := all_zero;
   for m := 0 to M-1 do
      m_adr[m] := V;

INVARIANTS
    v(0≤v<V):
   with virtual_map[v]
      loc = in iff mm[mm_adr] = vm[v] ∧
      loc = out iff sm[sm_adr] = vm[v] ∧
      loc = all_zero iff vm[v] = 0 ∧
      m_adr[i] = v iff vm[v] = mm[i] ∧
       m(0≤m<M)  (m ∉ m_pool) iff (∃ v such that
                  virtual_map[v].mmm_adr = m ∧
                  virtual_map[v].loc = in) ∧
       s(0≤s<S)  (s ∉ s_pool) iff (∃ v such that
                  virtual_map[v].sm_adr = s ∧
                  virtual_map[v].loc ≠ all_zero)

procedure mm_acquire (var m: mmpf);
begin
   if m_pool = empty then m_nonempty.wait;
   m := anyone of (m_pool);
   m_pool := m_pool - [m];
end mm_acquire;
```

Figure 3.5

Virtual_memory Specification Version 1

```
procedure mm_release (m: mmpf);
begin
   m_pool := m_pool + [m];
   m_nonempty.signal;
end mm_release;

procedure sm_acquire (var s: smpf);
begin
   if s_pool = empty then s_nonempty.wait;
   s := anyoneof(s_pool);
   s_pool := s_pool - [s];
end sm_acquire;

procedure sm_release (s: smpf);
begin
   s_pool := s_pool + [s];
   s_nonempty.signal;
end sm_release;
```

Figure 3.5 (continued)

Virtual_memory Specification Version 1

```
procedure fetch (va: virtual_address; var w: word);
begin
   if busy then monitorbusy.wait;
   busy := true;
   with virtual_map[va.vpno] do
      case loc of
         in: := mm[mm_adr, va.offset];
        out: begin
                mm_acquire(mm_adr);
                input(mm_adr, sm_adr);
                loc := in;
                m_adr[mm_adr] := va.vpno;
                w := mm[mm_adr, va.offset];
             end;
        all_zero: begin
                     mm_acquire(mm_adr);
                     sm_acquire(sm_adr);
                     mm[mm_adr] := 0;
                     loc := in;
                     m_adr[mm[adr] := va.vpno;
                     w := mm[mm_adr, va.offset];
                  end;
      end;
   busy := false;
   monitorbusy.signal;
end fetch;
```

Figure 3.5 (continued)

Virtual_memory Specification Version 1

```
procedure assign (va: virtual_address; w: word);
begin
   if busy then monitorbusy.wait;
   busy := true;
   with virtual_map[va.vpno] do
      case loc of
         in: mm[mm_adr, va.offset] := w;
         out: begin
                 mm_acquire(mm_adr);
                 input (mm_adr,sm_adr);
                 loc := in;
                 m_adr[mm_adr] := va.vpno;
                 mm[mm_adr, va.offset] := w;
              end;
         all_zero: begin
                      mm_acquire (mm_adr);
                      sm_acquire (sm_adr);
                      mm[mm_adr] := 0;
                      loc := in;
                      m_adr[mm_adr] := va.vpno;
                      mm[mm_adr, va.offset] := w;
                   end;
   end;
busy := false;
monitorbusy.signal;
end assign;
```

Figure 3.5 (continued)

Virtual_memory Specification Version 1

```
procedure clear_page;
begin
   if busy then monitorbusy.wait;
   with virtual_map[v] do
      case loc of
         in: begin
               m_adr[mm_adr] := V;
               loc := all_zero;
               mm_release(mm_adr);
               sm_release(sm_adr);
             end;
        out: begin
               loc := all_zero;
               sm_release(sm_adr);
             end;
       all_zero: {do nothing, possible error};
      end;
   monitorbusy.signal;
end clear_page;
```

Figure 3.5 (continued)

Virtual_memory Specification Version 1

```
procedure throwout (m:mmpf);
var v: 0..V;
begin
   v := m_addr[m];
   if (0≤v ∧ v<V) then
      begin
         output(m, virtual_map[v].sm_adr);
         m_adr[m] := V;
         virtual_map[v].loc := out;
         mm_release(m);
      end;
end throwout;
end virtual-memory;

process automatic_discard;
begin
   repeat
      for m := 0 to M-1 do
         begin
            wait_for_some_time;
            virtual_memory.throwout(m);
         end;
   forever;
end automatic_discard;
```

Figure 3.5 (continued)

Virtual_memory . pecification Version 1

function. The specification of these procedures is very much dependent upon the devices used by the system. The input/output procedures of the system can also be modelled as monitors [Hoare 1973]. We have assumed the behavior of the procedure input(m,s) to be copying the contents of the secondary memory page frame s into the main memory page frame m. This action is considered to be indivisible (the action can be made indivisible by the use of monitors). The behavior of the procedure output(m,s) is assumed to be copying the contents of the main memory page frame m into the secondary memory page frame s. The action of the procedure output is also considered to be indivisible. In actual fact these operations require a large time compared to the central processing unit's time to execute an instruction, and a process invoking these procedures will be delayed and the processor preempted. However, for the logical correctness of our specifications these procedures are considered to be indivisible and in practice are made to appear so.

With the current technology it takes a long time ($\approx$ 25 msec.) to input a page from secondary memory into the main memory compared to the time for a single main memory access ($\approx$ 1μsec.). Hence it is profitable to

introduce parallelism, so that if a process requires access to a virtual page which is currently in the secondary memory, we can allow other processes to access main memory while the required page is brought into the main memory.

We can do so by associating a condition variable per virtual page, and by blocking all access to that virtual page in the procedures fetch and assign if the page is in the process of being referenced. There is now no need for the boolean variable busy and the condition variable monitorbusy.

We will use the array of Boolean variables, page_inuse, and the array of condition variables, page_nonbusy, to block access to virtual pages in use. The modified procedures are given in figure 3.6.

3.4.4. Virtual_memory version 3: Separation of main memory access and page-fault handling

The solution presented in version 2 has the following disadvantages. When a process requires access to a virtual_page the monitor virtual_memory is entered. The mutual exclusion for the monitor may be implemented in two ways, with busy waiting or without busy waiting. Mutual exclusion with busy waiting can be implemented using the

```
page_nonbusy: array[0...V-1] of condition;
page_inuse: array[0..V-1] of Boolean;

procedure fetch (va: virtual_address; var w: word);
begin
   if page_inuse[va.vpno] then page_nonbusy[va.vpno].wait;
   page_inuse[va.vpno] := true;
   with virtual_map[va.vpno] do
      case loc of
         in: w := mm[mm_adr, va.offset];
         out: begin
                 mm_acquire(mm_adr);
                 input(mm_adr, sm_adr);
                 loc := in;
                 m_adr [mm_adr] := va.vpno;
                 w := mm[mm_adr, va.offset];
              end;
         all_zero: begin
                      mm_acquire(mm_adr);
                      sm_acquire(sm_adr);
                      mm[mm_adr] := 0;
                      loc := in;
                      m_adr[mm_adr] := va.vpno;
                      w := mm[mm_adr, va.offset];
                   end;
   end;
page_inuse[va.vpno] := false;
page_nonbusy[va.vpno].signal;
end fetch;
```

Figure 3.6

Virtual_memory Specification Version 2

```
procedure assign (va: virtual_address; w: word);
begin
   if page_inuse[va.vpno] then page_nonbusy[va.vpno].wait;
   page_inuse[va.vpno] := true;
   with virtual_map[va.vpno] do
      case loc of
         in: mm[mm_adr, va.offset] := w;
         out: begin
                 mm_acquire(mm_adr);
                 input(mm_adr, sm_adr);
                 loc := in;
                 m_adr[mm_adr] := va.vpno;
                 mm[mm_adr, va.offset] := w:
              end;
         all_zero: begin
                      mm_acquire(mm_adr);
                      sm_acquire(sm_adr);
                      mm[mm_adr] := 0;
                      loc := in;
                      m_adr[mm_adr] := va.vpno;
                      mm[mm_adr, va.offset] := w;
                   end;
      end;
      page_inuse[va.vpno] := false;
      page_nonbusy[va.vpno].signal;
end assign;
```

Figure 3.6 (continued)

Virtual_memory Specification Version 2

```
procedure clear_page(v: vpf);
begin
   if page_inuse[v] then page_nonbusy.wait;
   with virtual_map[v] do
      case loc of
         in: begin
                m_adr[mm_adr] := V;
                loc := all_zero;
                mm_release(mm_adr);
                sm_release(sm_adr);
             end;
         out: begin
                 loc := all_zero;
                 sm_release(sm_adr);
              end;
         all_zero: {do nothing, possible error};
      end;
   page_nonbusy[v].signal;
end clear_page;
```

Figure 3.6 (continued)

Virtual_memory Specification Version 2

hardware of level 0. When mutual exclusion with busy waiting is used, a flag is set to indicate that the monitor is busy, and a process desiring access to the monitor continuously checks the flag to see if it has been reset by the process currently in the monitor. Such an implementation of mutual exclusion results in wasted processor time. The implementation may be useful if the expected idle time is less than the processor time required for implementing mutual exclusion without busy waiting, using the enter and exit primitives of level 1.

In the case of the monitor virtual_memory we have conflicting needs. References to memory occur very often and most of the time the required virtual pages will be in the main memory. Thus, a process will not spend much time in the monitor; that is, processor idle time will not be large enough to justify implementing nonbusy waiting mutual exclusion. However, when a process requires access to a virtual page that is currently not in the main memory, the processor idle time (assuming shared pages, and current technology, which requires about 25 msec. to transfer a page from secondary memory to main memory) is expected to be much larger than that required for implementing mutual

exclusion without busy waiting. Since we can choose only one or the other method of implementing mutual exclusion, and neither is totally satisfactory, we have split up the monitor virtual_memory into two monitors, the monitor address_mapper and the monitor page_fault_handler. The monitor address_mapper performs operations on the virtual_ memory only if the contents are in the main memory, otherwise it signals a fault (page-fault) which is handled by the monitor page_fault_handler. The mutual exclusion for the monitor address_mapper is implemented with busy waiting since it is expected that a process will be inside the monitor for only a short time and calls will be very frequent. The mutual exclusion for the monitor page_fault_ handler is implemented without busy waiting, since it is expected that a process will spend much time inside the monitor if it accesses it.

We will also need two re-entrant procesures, fetch and assign, that will be called by the user processes. These procedures will in turn call the procedures mm_fetch and mm_assign of the monitor address_mapper and in case of a page-fault they will call the procedure bringin of the monitor page_fault_handler. The use of these procedures makes the implementation of the virtual memory invisible

to the user processes. The definition of these procedures is given in figure 3.7. Note that in case of a page-fault, after the page has been brought into the main memory, the procedure mm_fetch or mm_assign is re-invoked to try the operation again. The process of discarding pages from the main memory is asynchronous with the attempts at accessing the page and it is possible that an attempt to access a page may never succeed. Any solution which guarantees that a virtual page can be accessed by a process in a finite number of attempts will increase overhead. If in practice it is found that a process is unable to access a virtual page (which seems to be a highly unlikely event) in a reasonable number of attempts, then the solutions that involve locking a page in the main memory may be devised. These solutions can guarantee access to a virtual page in a reasonable number of attempts. It is assumed that by appropriately decreasing the rate of discarding pages the possibility that a process fails to access a page in more than one attempt can be made negligible.

The monitors address_mapper and page_fault_handler, and the procedures fetch and assign are given in figure 3.7.

```
monitor address_mapper;
begin
   var vmap: array[0..V-1] of 0..M;
      {initialization},
   for v := 0 to V-1 do
   INVARIANT
    m(0≤m<M) ∧  v(0≤v<V): v_map[v] = m => (vm[v] = mm[m])
   procedure mm_fetch (va: virtual_address; var w:word;
                       var pf: Boolean);
   var m: 0..M;
   begin
      m := vmap[va.vpno];
      pf := false;
      if m ≠ M then w := mm[m,va.offset];
      else pf := true;
   end·mm_fetch;

   procedure mm_assign (va: virtual_address; w:word;
                        var pf: Boolean);
   var m: 0..M;
   begin
      m := vmap[va.vpno];
      pf := false;
      if m ≠ M then mm[m,va.offset] := w
      else pf := true;
  end mm_assign;

  procedure set_map (v: vpf; m: 0..M);
  begin
     vmap[v] := m;
  end set_map;
  vmap[v] := M;
end address_mapper;
```

Figure 3.7

Virtual_memory Specification Version 3

```
monitor page_fault_handler;
begin
   type vp_state = record loc(in, out, all_zero);
                          mm_adr: 0..M-1; sm_addr: 0..S-1 end;
   var virtual_map: array[0..V-1] of vp_state;
      m_adr: array[0..M-1] of 0..V;
      m_pool: set of mmpf;
      s_pool: set of smpf;
      page_nonbusy: array[0..V-1] of condition;
      page_inuse: array[0..V-1] of Boolean;
      m_nonempty, s_nonempty: condition;

{Initially}
   m_pool := [0.. M-1];
   s_pool := [0..S-1];
   busy := false;
   for v := 0 to V-1 do
      virtual_map[v].loc := all_zero;
   for m := 0 to M-1 do
      m_adr[m]  := V;

INVARIANTS
   ∀v(0≤v<V):
   with virtual_map[v]
      loc = in iff mm[mm_adr] = vm[v] ∧
      loc = out iff sm[sm_adr] = vm[v] ∧
      loc = all_zero iff vm[v] = 0 ∧
      m_adr[i] = v iff vm[v] = mm[i] ∧
   ∀m(0≤m<M)  (m ∈ m_pool) iff (∃v such that
                                virtual_map[v].mm_adr = m ∧
                                virtual_map[v].loc = in) ∧
   ∀s(0≤s<S)  (s ∈ s_pool) iff (∃v such that
                                virtual_map[v].sm_addr = s ∧
                                virtual_map[v].loc ≠ all_zero)
```

Figure 3.7 (continued)

```
procedure mm_acquire (var m: mmpf);
begin
   if m_pool = empty then m_nonempty.wait;
   m := anyoneof(m_pool);
   m_pool := m_pool - [m];
end mm_acquire;

procedure mm_release (m: mmpf);
begin
   m_pool := m_pool + [m];
   m_nonempty.signal;
end mm_release;

procedure sm_acquire (var s: smpf);
begin
   if s_pool = empty then s_nonempty.wait;
   s := anyoneof(s_pool);
   s_pool := s_pool - [s];
end sm_acquire;

procedure sm_release (s: smpf);
begin
   s_pool := s_pool + [s];
   s_nonempty.signal;
end sm_release;
```

Figure 3.7 (continued)

Virtual_memory Specification Version 3

```
procedure bringin (v: vpf);
begin
   if page_inuse[v] then page_nonbusy[v].wait;
   page_inuse[v] := true;
   with virtual_map[v] do
      case loc of
         in: {do nothing};
         out: begin
                 mm_acquire(mm_adr);
                 input(mm_adr, sm_adr);
                 loc := in;
                 m_adr[mm_adr] := v;
                 address_mapper.set_map(v,mm_adr);
              end;
         all_zero: begin
                      mm_acquire(mm_adr);
                      sm_acquire(sm_adr);
                      mm[mm_adr] := 0;
                      loc := in;
                      m_adr[mm_adr] := v;
                      address_mapper.set_map(v,mm_adr);
                   end;
      end;
   page_inuse[v] := false;
   page_nonbusy[v].signal;
end bringin;
```

Figure 3.7 (continued)

Virtual_memory Specification Version 3

```
procedure clear_page(v: vpf);
begin
   if page_inuse[v] then page_nonbusy.wait;
   with virtual_map[v] do
      case loc of
         in: begin
                address_mapper.set_map(v,M);
                m_adr[mm_adr] := V;
                loc := all_zero;
                mm_release(mm_adr);
                sm_release(sm_adr);
             end;
        out: begin
                loc := all_zero;
                sm_release(sm_adr);
             end;
        all_zero: {do nothing, possible error};
      end;
   page_nonbusy[v].signal;
end clear_page;

procedure throwout (m: mmpf);
var v: 0..V;
begin
   v := m_adr[m];
   if (0≤v ∧ v<V) then
      begin
         address_mapper.set_map(v,M);
         output(m,  virtual_map[v].sm_adr);
         m_adr[m] := V;
         virtual_map[v].loc := out;
         mm_release(m);
      end;
end throwout;
end page_fault_handler;
```

Figure 3.7 (continued)

Virtual_memory Specification Version 3

```
procedure fetch (va: virtual_address; var w: word);
var pf: Boolean;
begin
   address_mapper.mm_fetch(va,w,pf);
   while pf do
      begin
         page_fault_handler.bringin(va.vpno);
         address_mapper.mm_fetch(va,w,pf);
      end;
end fetch;

procedure assign (va: virtual_address; w: word);
var pf: Boolean;
begin
   address_mapper.mm_assign(va,w,pf);
   while pf do
      begin
         page_fault_handler.bringin(va.vpno);
         address_mapper.mm_assign(va,w,pf);
      end;
end assign;
```

Figure 3.7 (continued)

Virtual_memory Specification Version 3

3.4.5. Virtual-memory version 4: Minimization of address-mapper space requirements

There are several points to be noted about our previous solution. Although there are now two monitors, there is a relationship between the data structures of the two monitors. The correctness of the solution depends upon this relationship. To satisfy the invariants of the original solution, we need the following invariants.

```
(vmap[v] = m) iff (vm[v] = mm[m])
                  iff (virtual_map[v].loc = in ∧
                       virtual_map[v].mm[adr = m)
```

To maintain the inter_monitor invariant relation, we need the constraint that the relevant data structures be modified only as a result of a call from the other monitor. In our case the procedure set_map of the monitor address_mapper will be called only from within the monitor page_fault_handler. We do not have a syntactical construct to enforce this constraint, but the inter_monitor invariant relation is necessary for the correctness of the solution. Note that the invariants for each of the monitors have to be satisfied individually at the initiation and termination of a call from outside the monitor to one of its procedures (or functions). The inter_monitor invariant is the required output assertion of the procedure

bringin of the monitor page_fault_handler.

We observe that in the previous solution, the monitor address_mapper is expected to have processes within it for a very short time. It satisfies this condition but requires a very large amount of storage space. As specified, we need to provide an entry in the array vmap for each virtual page, including all_zero pages and pages that are currently stored in secondary memory. Since the number of such pages is very large, the space required for the array vmap will be too large to be economical. We really need to know the mapping from a virtual address to a real address in this monitor only if the contents of the relevant virtual page are currently in the main memory. Thus, trading memory space for execution time we can construct an inverse map that maps every main memory page frame into a virtual page number if and only if the main memory page frame contains the contents of the virtual page, otherwise the main memory page frame will be mapped into the default, and out of range, value V.

The modified monitor address_mapper is given in figure 3.8. The procedure adr_trans searches through the map,adr_map, to find the identity of the main memory page frame that contains the relevant virtual page, if any.

```
monitor address_mapper;
begin
var m_adr: array [mmpf] of 0..V;
INVARIANT:  v(0≤v<V) (m_adr[m] = v) iff (vm[v] = mm[m])

procedure mm_fetch (va: virtual_address; var w:word;
                    var pf: Boolean);
var m: 0..M;
begin
   adr_trans(va.vpno,m,pf);
   if ¬ pf    then w := mm[m,va.offset];
end mm_tetch;

procedure mm_assign(va: virtual_address; w:word;
                    var pf: Boolean);
var m: 0..M;
begin
   adr_trans(va.vpno,m,pf);
   if ¬ pf    then mm[m,va.offset] := w;
end mm_assign;

procedure adr_trans (v: vpf; var m: 0..M; var pf: Boolean);
begin
   m := 0;
   while m < M ∧    (m_adr[m] ≠ v) do
      m := m + 1;
   pf := (m=M);
end adr_trans;

procedure set_map (v: 0..V; m: mmpf);
begin
   m_adr[m] := v;
end set_map;

{initially};
for m := 0 to M-1 do
   m_adr[m] := V;
end address_mapper;
```

Figure 3.8

Virtual_memory Specification Version 4

If there is no main memory page frame that contains the relevant virtual page the procedure adr_trans will return the value of the Boolean variable pf as true, otherwise pf is returned as false. To speed up the searching process, the map adr_map, may be implemented as an associative map.

3.4.6. Virtual-memory version 3.4.5: Parallel input/output

Let us consider the performance of the monitor page_fault_handler. Recall that the procedures input and output are considered to be indivisible operations (and assumed to be specified externally). Hence, when one of these procedures is invoked from within the monitor page_fault_handler, access to the monitor is blocked until the operation is completed, and so only one page-fault can be handled at a time. The input/output operation is a time-consuming one, and the current secondary memory devices (e.g. sectored drums) can profitably use parallelism. So it is advantageous to transform our solution such that we can allow multiple requests for input/output to be made.

The reason we are prevented from making multiple requests for input is that when the first request is made, the monitor page_fault_handler is blocked (access is

not released) till the operation is completed. If we could release access to the monitor for other processes requiring access to other virtual pages, then these processes could also make input/output requests. Since the procedure input is considered to be indivisible, the only way to release access to the monitor is to invoke the procedure from outside the monitor. When a process suffers a page-fault it enters the monitor page_fault_handler invoking the procedure bringin. If it is found necessary to invoke the procedure input, access to the monitor is released, the procedure input is invoked, and then the monitor is re-entered to continue the processing. We have to take care that no other process requiring access to the same virtual page enters the monitor, or if it does then it is blocked just after entering the monitor. Thus, what we need is a procedure that will return a value that indicates if input is required and the actual parameters for the call to the procedure input. The procedure should also lock up access to the virtual page if input is required. We call this procedure lock_for_input. Similarly, we will also need a procedure to release access in case output is required and will call that procedure lock_for_-output. We will also need to modify the procedures

bringin and throwout, as well as the procedures fetch and assign and the process automatic discard. The modified procedures are given in figures 3.4.9a to 3.4.9e. Note that the correctness is preserved because exclusive access on the concerned virtual page is still maintained and only access to other virtual pages is allowed. Since the status of virtual pages is independent of each other, operating on one virtual page does not affect another virtual page and so our transformation of the solution preserves the indivisible property provided by the monitor page_fault_-handler for each virtual page.

We will introduce a new re-entrant procedure, page_-fault, that will be invoked by the re-entrant procedures fetch and assign when a page-fault is discovered. The procedure page_fault hides the transformations of the monitor page_fault_handler procedures from the procedures fetch and assign. The definition of the procedure page_-fault is given in figure 3.4.9b.

If we had chosen to specify the input/output procedures, then we could have affected the above transformation in another way. The input/output operation takes a long time to complete, mostly because of the time spent in waiting for the device to be free (including latency time)

```
procedure lock_for_input (v: vpf; var m: mmpf; var s: smpf;
                          var p_out: Boolean);
begin
   if page_inuse[v] then page_nonbusy[v].wait;
   p_out := false;
   with virtual_map[v] do
      case loc of
         in: page_nonbusy[v].signal;
         out: begin
                 mm_acquire(mm_adr);
                 m := mm_adr;
                 s := sm_adr;
                 p_out := true;
                 page_inuse[v] := true;
              end;
         all_zero: begin
                      mm_acquire(mm_adr);
                      sm_acquire(sm_adr);
                      mm[mm_adr] := 0;
                      loc := in;
                      m_adr[mm_adr] := v;
                      address_mapper.set_map(v,mm_adr);
                      page_nonbusy[v].signal;
                   end;
      end;
end lock_for_input;

procedure bringin (v:vpf);
begin
   with virtual_map[v] do
      if loc = out then
         begin
            loc := in;
            m_adr[mm_adr] := v;
            address_mapper.set_map(v,mm_adr);
            page_inuse[v] := false;
            page_nonbusy[v].signal;
         end;
      else ERROR;
end bringin;
```

(a)

Figure 3.9

Virtual_memory Specification Version 5

```
procedure lock_for_output (m: mmpf; var s: smpf;
                           var vp_out: Boolean);
var v: 0..V;
begin
   vp_out := false;
   v := m_adr[m];
   if (0≤v ∧ v<V) then
      begin
         page_inuse[v] := true;
         address_mapper.set_map(V,m);
         vp_out := true;
         s := virtual_map[v].sm_adr;
      end;
end lock_for_output;

procedure throwout (m: mmpf);
var v: 0..V-1;
begin
   v := m_adr[m];
   m_adr[m] := V;
   virtual_map[v].loc := out;
   mm_release(m);
   page_inuse[v] := false;
   page_nonbusy[v].signal;
end throwout;
```

(b)

Figure 3.9 (continued)

Virtual_memory Specification Version 5

```
procedure page_fault (v: vpf);
var m:mmpf; s: smpf; out: Boolean;
begin
   page_fault_handler.lock_for_input(v,m,s,out);
   if out then
      begin
         input(m,s);
         page_fault_handler.bringin(v);
      end;
end page_fault;
```

(c)

```
procedure fetch (va: virtual_address; var w:word);
var pf: Boolean;
begin
   address_mapper.mm_fetch(va,w,pf);
   while pf do
      begin
         page_fault (va.vpno);
         address_mapper.mm_fetch(va,w,pf);
      end;
end fetch;

procedure assign (va: virtual_address; w: word);
var pf: Boolean;
begin
   address_mapper.mm_assign(va,w,pf);
   while pf do
      begin
         page_fault (va.vpno);
         address_mapper.mm_assign(va,w,pf);
      end;
end assign;
```

(d)

Figure 3.9 (continued)

Virtual_memory Specification Version 5

```
process automatic_discarder;'
var m: 0..M-1;
    s: 0..S-1;
    p-output: Boolean;

begin
   repeat
      for m := 0 until M-1 do
         begin
            wait_for_some_time;
            page_fault_handler.lock_for_output(m,s,p-output);
            if p-output then
               begin
                  output(m,s);
                  page_fault_handler.throwout(m,s);
               end;
         end;
   forever;
end automatic_discarder;
```

(e)

Figure 3.9 (continued)

Virtual_memory Specification Version 5

and in the transmission of information. In the specification of the input/output procedures, a process is suspended after its request is queued until the requested operation is completed. By including the monitor containing the input/output procedures as part of the monitor page_fault_handler, we would be able to release access to the monitor page_fault_handler after the request is queued, thus gaining the ability to queue multiple input/-output requests.

The above approach seems to us to be less desirable than the one we adopted. It makes the monitor page_fault_-handler more complex. It has been suggested that monitors be viewed as abstract data structures [Hoare 1974, Brinch Hansen 1975] with well defined properties. By combining essentially separate abstractions into one abstraction we complicate the task of defining the abstract properties. On the other hand, the approach we have adopted maintains the identity of separate monitors that can each be given separate abstract properties. Our approach has the disadvantage of requiring extra procedure definitions and a flow of control that is more complex than that needed for the above approach. The flow of control is more complex because we have to leave the monitor after inspecting its

state and then re-enter the monitor after performing an operation outside the monitor to record the effect of the operation in the variables inside the monitor.

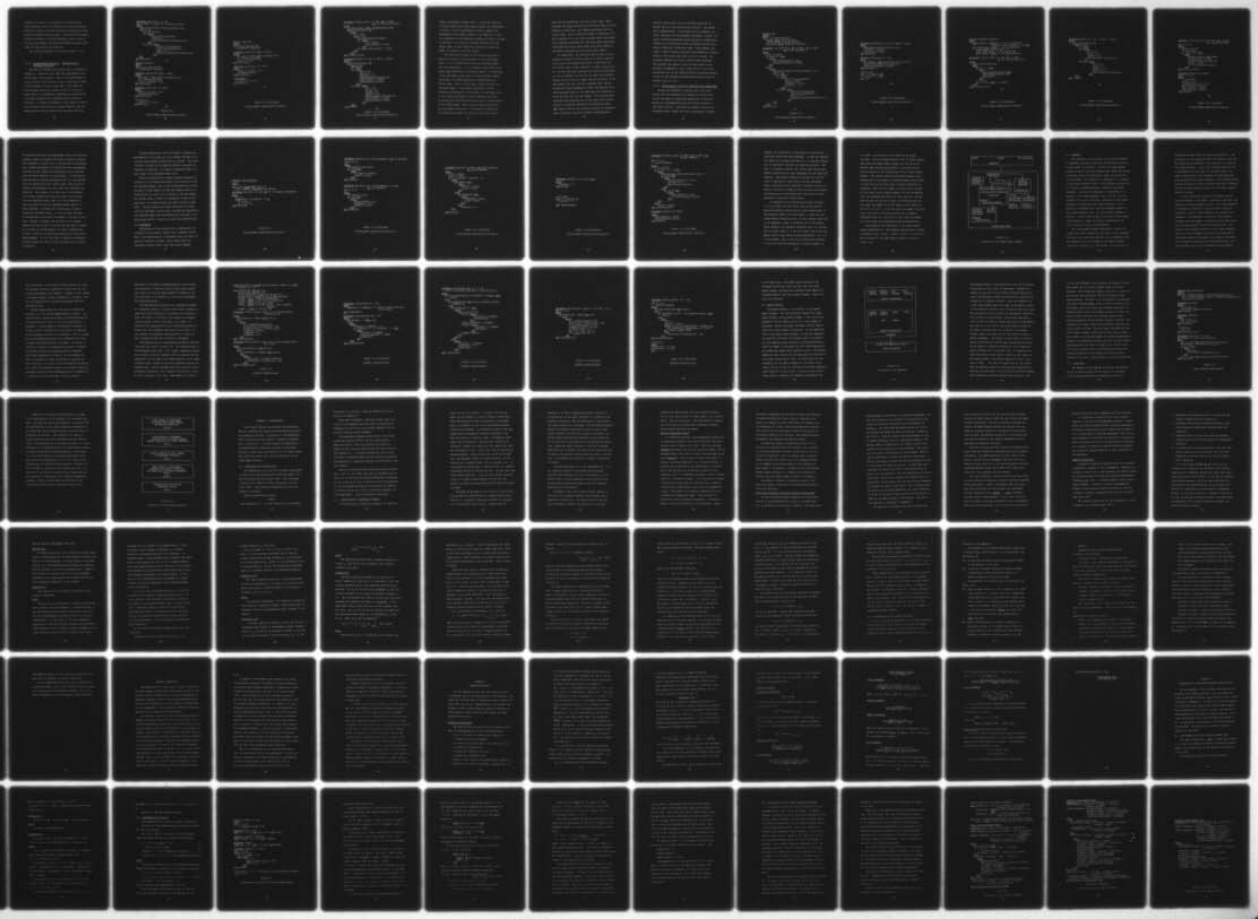
3.4.7. Virtual-memory version 6: Separate resource allocation monitors.

In keeping with the above discussion, to simplify the invariants for the monitor page_fault_handler, we will separate the resource allocation procedures mm_acquire, mm_release, sm_acquire, and sm_release from the monitor page_fault_handler into separate monitors. The monitor mm_alloc will consist of the procedures mm_acquire and mm_release. The monitor sm_alloc will consist of the procedures sm_acquire and sm_release. The resources will now be acquired through the procedure page_fault when input is required. The procedure lock_for_input is now modified to lock the page only if the page is not in the main memory.

The transformation preserves the correctness, because the actions that are now being transferred to the procedure page_fault were performed with access to the concerned virtual page locked and this condition is still satisfied. Further, when the resource allocation procedures were invoked from within the monitor it was

AD-A040 699 STANFORD UNIV CALIF STANFORD ELECTRONICS LABS F/G 9/2
A VERIFIED SPECIFICATION OF A HIERARCHICAL OPERATING SYSTEM.(U)
JAN 76 A R SAXENA N00014-75-C-0601
UNCLASSIFIED TR-107 NL

2 OF 3
AD A040699



possible for access to the monitor to be released for other processes, and this condition is also satisfied now because of the exit from the monitor on termination of the modified procedure lock_for_input. Note that the procedure bringin is modified to unlock the page in case it was an all_zero page because the modified procedure lock_for_input locks the page under this condition.

The modified procedures are given in figure 3.10.

3.4.8. Virtual-memory version 7: Minimization of condition variables.

We have now obtained the solution that is verified in appendix C. There are still some more improvements that can be made to the solution. One of the disadvantages of the solution we have obtained is that it requires a conditional variable for each virtual page. The number of virtual pages can be very large; in fact, it may be so large that it is economically undesirable to allocate the main memory necessary for implementing the condition variables. A scheme to implement a large number of condition variables (and monitors) trading execution time for memory space has been proposed that can reduce the main

```
procedure page_fault (v: vpf);
var m: mmpf; s: smpf; st: (in,out,all_zero);
begin
   page_fault_handler.lock_for_input(v,s,st);
   case st of
      in: {do nothing};
      out: begin
              mm_alloc.mm_acquire(m);
              input(m,s);
              page_fault_handler.bringin(v,m,s);
           end;
      all_zero: begin
                   mm_alloc.mm_acquire(m);
                   sm_alloc.sm_acquire(s);
                   mm[m] := 0;
                   page_fault_handler.bringin(v,m,s);
                end;
   end;
end page_fault;
```

(a)

```
monitor mm_alloc;
begin
var m_pool: set of mmpf;
   m_nonempty : condition;

procedure mm_acquire (var m: mmpf);
begin
   if m_pool = empty then m_nonempty.wait;
   m := anyoneof(m_pool);
   m_pool := m_pool - [m];
end mm_acquire;

procedure mm_release (m: mmpf);
begin
   m_pool := m_pool + [m];
   m_nonempty.release;
end mm_release;

{initially};
   m_pool := [0..M-1];
end mm_alloc;
```

(b)

Figure 3.10

Virtual_memory Specification Version 6

```
monitor sm_alloc;
begin
var s_pool: set of smpf;
   s_nonempty: condition;

procedure sm_acquire (var s: smpf);
begin
   if s_pool = empty then s_nonempty.wait;
   s := anyoneof(s_pool);
   s_pool := s_pool - [s];
end sm_acquire;

procedure sm_release (s: smpf);
begin
   s_pool := s_pool + [s];
   s_nonempty.signal;
end sm_release;

{initially};
   s_pool := [0..S-1];
end sm_alloc;
```

(c)

Figure 3.10 (continued)

Virtual_memory Specification Version 6

```
procedure lock_for_input (v: vpf; var s: smpf;
                          var st: (in,out,all_zero);
begin
   if page_inuse[v] then page_nonbusy[v].wait;
   with virtual_map[v] do
      begin
         st := loc;
         case loc of
            in: page_nonbusy[v].signal;
            out: begin
                    s := sm_adr;
                    page_inuse[v] := true;
                 end;
            all_zero: page_inuse[v] := true;
         end;
      end;
end lock_for_input;

procedure bringin (v: vpf; m: mmpf; s: smpf);
begin
   with virtual_map[v] do
      case loc of
      in: ERROR;
      out: begin
              loc := in;
              mm_adr := m;
              mm_adr[m] := v;
              address_mapper.set_map(v,m);
              page_inuse[v] := false;
              page_nonbusy[v].signal;
           end;
      all_zero: begin
                   loc := in;
                   sm_adr := s;
                   mm_adr := m;
                   m_adr[m] := v;
                   address_mapper.set_map(v,m);
                   page_inuse[v] := false;
                   page_nonbusy[v].signal;
                end;
      end;
end bringin;
```

(d)

Figure 3.10 (continued)

memory requirements [Saxena 1975]. Since the condition variables used by the simple memory manager are implemented at level 1, and the performance of level 1 affects the performance of all higher levels, it is important to make the operations on the condition variables at level 1 as fast as possible. So, instead of trading execution time for memory space, we will modify our solution to reduce the number of condition variables needed.

The condition variables in the monitor page_fault_-handler are used to lock up access to the virtual page when: 1) it is being written into the secondary memory from the main memory; 2) it is being written into a main memory page frame from the secondary memory; 3) resources, a free main memory page frame or a free secondary memory page frame, are being acquired for transferring the virtual page. When a virtual page is being written into secondary memory, a main memory page frame, the one currently containing the virtual page, is associated with it and it may be possible to use a condition variable associated with the main memory page frame to block access to the virtual page. When a page is being written into the main memory then also it is possible to associate a main memory page frame, the one into which the virtual

page will be transferred, with the virtual page. When resources are being acquired for the virtual page it is not possible to associate a main memory page frame with the virtual page. We can reduce the number of condition variables required if we can find a solution which uses condition variables associated with the main memory page frame containing the virtual page rather than using condition variables associated with the virtual page itself.

As noted above, the only time that we need to use a condition variable to block access to a virtual page and do not have a main memory page frame to associate with the virtual page is when resources are being acquired for it. We know that when resources are being acquired there will be no attempts to write out the page into secondary memory (we only write out pages that are currently in the main memory) and that the only processes that need be blocked are those attempting to clear the page and those suffering page faults on the same page and attempting to bring the page into the main memory. We can block these processes from accessing the virtual page while another process is acquiring resources for the page by making them as well as the process acquiring the resources enter the monitor page_fault_handler through another

monitor, monitor pfh, and not releasing access to the monitor pfh till the resources are acquired. The monitor pfh is defined below. It consists of the procedures initiate, terminate and the procedure zero_page. Further, instead of blocking processes inside the monitor page_fault_handler we can block them inside the monitor pfh by introducing a condition variable per mmpf. The procedure initiate is used to acquire resources when needed and to block access to the virtual page that is still in input. The procedure terminate is used to unblock these processes. The procedure zero_page is used to block access to the processes attempting to clear a page while it is in use. The monitor pfh and the modified monitor page_fault_handler as well as the modified procedure page_fault are given in figures 3.11a to 3.11c.

3.4.9. Virtual_memory version 8: Modified page replacement

We have now obtained a solution that is not only correct but also efficient (as compared to earlier solutions) and simple to understand (especially if one goes through the transformations from the initial solution to the final solution). Note that we assumed a certain replacement policy, namely the cyclic replacement of pages.

```
monitor pfh;
begin
var st: (in,out,all_zero);
   mm_map: array [vpf] of mmpf;
   locked: array [vpf] of Boolean;
   page_free: array [mmpf] of condition;

procedure initiate(v:vpf; var m: mmpf; var s: smpf;
                   var p_out: Boolean);
begin
   p_out := false;
   if locked[v] then
      begin
         page_free[mm_map[v]].wait;
         page_free[mm_map[v]].signal;
      end
   else
      begin
         page_fault_handler.lock_for_input(v,s,st);
         case st of
            in: ERROR;
            out: begin
                    mm_alloc.mm_acquire(m);
                    p_out := true;
                    locked[v] := true;
                    mm_map[v] := m;
                 end;
            all_zero: begin
                         mm_alloc.mm_acquire(m);
                         sm_alloc.sm_acquire(s);
                         mm[m] := 0;
                         page_fault_handler.bringin(v,m,s);
                      end;
         end;
   end;
end initiate;
```

(a)

Figure 3.11

Virtual_memory Specification Version 7

```
procedure terminate (v: vpf; m: mmpf; s: smpf);
begin
   page_fault_handler.bringin(v,m,s);
   locked[v] := false;
   page_free[m].signal;
end terminate;

procedure zero_page (v: vpf);
begin
   if locked[v] then page_free[mm_map[v]].wait;
   page_fault_handler.clear_page(v);
   page_free[mm_map[v]].signal;
end zero_page;

{initially};
for v := 0 to V-1 do
   locked[v] := false;
end pfh;
```

(a)

Figure 3.11 (continued)

Virtual_memory Specification Version 7

```
monitor page_fault_handler;
begin
   type vp_state = record loc: (in,out,all_zero);
                          mm_adr: 0..M-1; sm_adr: 0..S-1 end;
   var virtual_map: array [0..V-1] of vp_state;
      m_adr: array [0..M-1] of 0..V;
      page_inuse: array [0..M-1] of Boolean;
      page_nonbusy: array [0..M-1] of condition;

procedure lock_for_input (v: vpf; var s: smpf;
                          var st: (in,out,all_zero);
begin
   with virtual_map[v] do
   begin
      if loc = in then
         begin
            if page_inuse[mm_adr] then
            page_nonbusy[mm_adr].wait
            else ERROR;
         end;
      st := loc;
      if loc = out then s := sm_adr;
   end;
end lock_for_input;
```

(b)

Figure 3.11 (continued)

Virtual_memory Specification Version 7

```
procedure bringin (v: vpf; m: mmpf; s: smpf);
begin
   with virtual_map[v] do
      case loc of
         in: ERROR:
         out: begin
                 loc := in;
                 mm_adr := m;
                 m_adr[m] := v;
                 address_mapper.set_map(v,m);
              end;
         all_zero: begin
                      loc := in;
                      sm_adr := s;
                      mm_adr := m;
                      m_adr[m] := v;
                      address_mapper.set_map(v,m);
                   end;
      end;
end bringin;
```

(b)

Figure 3.11 (continued)

Virtual_memory Specification Version 7

```
procedure lock_for_output (m: mmpf; var s: smpf;
                           var vp_out: Boolean);
var v: 0..V;
begin
   vp_out := false;
   v := m_adr[m];
   if (0≤v ∧ v<V) then
      begin
         if page_inuse[m] then ERROR;
         page_inuse[m] := true;
         address_mapper.set_map(v,m);
         vp_out := true;
         s := virtual_map[v].sm_adr;
      end;
end lock_for_output;

procedure throwout (m: mmpf);
var v: 0..V-1;
begin
   v := m_adr[m];
   m_adr[m] := V;
   virtual_map[v].loc := out;
   mm_alloc.mm_release(m);
   page_inuse[m] := false;
   page_nonbusy[m].signal;
end throwout;
```

(b)

Figure 3.11 (continued)

```
procedure clear_page (v: vpf);

begin
   with virtual_map[v] do
   begin
      if loc = in ∧ page_inuse[mm_adr] then
      page_nonbusy[mm_adr].wait;
      case loc of
         in: begin
                address_mapper.set_map(v,m);
                m_adr[mm_adr] := V;
                mm_alloc.mm_release(mm_adr);
                sm_alloc.sm_release(sm_adr);
                loc := all_zero;
             end;
        out: begin
                sm_alloc.sm_release(sm_adr);
                loc := all_zero;
             end;
        all_zero: {do nothing};
     end;
end clear_page;

{initially}
for v := 0 to V-1 do
   virtual_map[v].loc := all_zero;
for m := 0 to M-1 do
   begin
      m_adr[m] := V;
      page_inuse[m] := false;
   end;
end page_fault_handler;
```

(b)

Figure 3.11 (continued)

Virtual_memory Specification Version 7

```
procedure page_fault (v: vpf);
var m: mmpf; s: smpf; p_out: Boolean;
begin
   pfh.initiate(m,s,p_out);
   if p_out then
      begin
         input(m,s);
         pfh.terminate(v,m,s);
      end;
end page_fault;
```

(c)

Figure 3.11 (continued)

Virtual_memory Specification Version 7

We stated earlier that the replacement policy can be easily changed, simply by changing the process automatic discard. The statement is correct only if the policies to be adopted (e.g. random replacement or first-in-first-out replacement policies) do not require any information that is derived from the actual behavior of the programs. If the replacement policies do require information that is to be derived from the operations on the virtual pages, then we have to specify the procedures that will gather the necessary information. For example, if we know that a virtual page has not been changed since the last time it was written into the secondary memory, then it is not necessary to write the page out into the secondary memory when it is being replaced. To gather this information, we have to modify the procedure assign_1 so that it marks the page as changed when a new value is assigned to any word in the page. Further, to ensure that the page is not changed between the time we test it and the time the page is marked as not being in the main memory, we have to perform both of these actions as an indivisible operation of the monitor address_mapper. We will also need to modify the procedure lock_for_output such that it locks the page only if output is required.

Another modification that we can make to improve the performance of the system is to not replace the page if it has been used recently (either read or written). This modification is based on the observed behavior (principle of locality) of programs: If a page is referenced then it is very likely to be referenced again soon.

These two modifications suggest another modification. If page is to be replaced and needs to be written out into the secondary memory, then it may be desirable not to mark the page as being absent from the main memory while it is being written out and to let it remain in the main memory for another cycle, so that if a reference is made to the page while it is being written out there will be no page-fault. The main memory page frame occupied by the page would not have been released until the page was copied into the secondary memory and the modification will help in reducing page traffic. Figure 3.12 gives the modifications.

3.10 Discussion

The purpose of this exercise was to demonstrate the usefulness of the monitor concept and of stepwise refinement in the specification of procedures that are useful in practical operating systems. Hoare [Hoare 1973] has described a paging system using the monitor concept.

```
monitor address_mapper;
begin
var m_adr: array[mmpf] of 0..V;
   used, changed: array[mmpf] of Boolean;
procedure adr_trans (v: vpf; var m: 0..M; var pf: Boolean);
begin
   m := 0;
   while m<M ∧ ¬(m_adr[m] = v) do
      m := m+1;
     pf := (m = M);
end adr_trans;
```

Figure 3.12

Virtual_memory Specification Version 8

```
procedure mm_fetch (va: virtual_address; var pf: Boolean);
var m: 0..M;
begin
   adr_trans(va.vpno,m,pf);
   if ¬pf then
      begin
         w := mm[m,va.offset];
         used[m] := true;
      end;
end mm_fetch;


procedure mm_assign (va: virtual_address; w: word;
                     var pf: Boolean);
var m: 0..M;
begin
   adr_trans(va,m,pf);
   if ¬pf then
      begin
         mm[m,va.offset] := w;
         changed[m] := true;
         used[m] := true;
      end;
end mm_assign;
```

Figure 3.12 (continued)

Virtual_memory Specification Version 8

```
procedure mark_out (m: mmpf; var m_out: Boolean;
                    var w_out: Boolean);
begin
   m_out := true;
   w_out := false;
   if (0 ≤ m_adr[m] ∧ m_adr[m] < V)
      begin
         if used[m] then
            begin
               used[m] := false;
               m_out := false;
            end;
         else
            if changed[m] then
               begin
                  changed[m]:= false;
                  w_out := true;
               end
            else
               m_adr[m] := V;
      end;
end mark_out;
```

Figure 3.12 (continued)

Virtual_memory Specification Version 8

```
procedure set_map (v: 0..V; m: mmpf);
begin
   m_adr[m] := v;
end set_map;



{initially}
for m := 0 to M-1 do
   m_adr[m] := V;

end address_mapper;
```

Figure 3.12 (continued)

Virtual_memory Specification Version 8

```
procedure lock_for_output (m: mmpf; var s: smpf; var
                           vp_out: Boolean);

var v: 0..V;
    m_out: Boolean;

begin
   v := m_adr[m];
   if (0 ≤ v ∧ v < V) then
      begin
         address_mapper.mark_out(m,m_out,vp_out);
         if m_out then
            begin
               virtual_map[v].loc := out;
               m_adr[m] := V;
               mm_alloc.mm_release(m);
            end
         else
            if vp_out then
               begin
                  page_inuse[m] := true;
                  s := virtual_map[v].sm_adr;
               end;
      end
   else
      vp_out := false;
end lock_for_output;

procedure throwout (m: mmpf);

begin
   page_inuse[m] := false;
   page_nonbusy[m].signal;
end throwout;
```

Figure 3.12 (continued)

Virtual_memory Specification Version 8

However, his description is very similar to our earlier solutions and has the same drawbacks. It does not separate the access to a virtual page when it is in the main memory, and the page-fault handling into separate monitors. Further, it requires a monitor per virtual page and the procedures responsible for page replacement are not explained properly. They access condition variables that are not local to the monitor within which they are specified. Hoare's description also specifies a process as part of a monitor, and for his scheme to work correctly the discarding process has to be modified and there may be as many processes as potential virtual pages.

An inspection of the specification given in figure 3.12 will reveal that the monitor pfh and the monitor page_fault_handler require storage space proportional to the potential number of virtual pages, V. Note that the simple memory manager provides a virtual address space only to the scheduler (level 3) processes and if the address space needed by the scheduler processes (that is, the number of virtual pages V) is not very large, then we can use memory space to meet these storage requirements. If V is a large number, then it may not be economically feasible to store the required information in the main memory at

all times. In this case we can adopt the following strategy. We can introduce another level of memory management below the simple memory manager for the use of the monitors pfh and page_fault_handler. The specification will be similar to the specification of the simple memory manager. The monitor sm_alloc which needs address space proportional to the number of secondary memory page frames, S, could also use the same level of memory management if necessary. Note that the number of virtual pages managed by this lower level of memory manager will be considerably less, for assuming that there are 1000 lines in a virtual page and that we require 10 lines of information per virtual page, the number of virtual pages at this level will be 1% of V (0.01 V). If we find even this number to be too large, then we can introduce another level of memory management below it. Note that at each level the number of virtual pages to be managed decreases exponentially.

This completes the discussion of the simple memory manager specification. The complete specification is given in figures 3.9(d), 3.9(e), 3.10(b), 3.10(c), 3.11 and 3.12. The structure of the simple memory manager is given in figure 3.13.

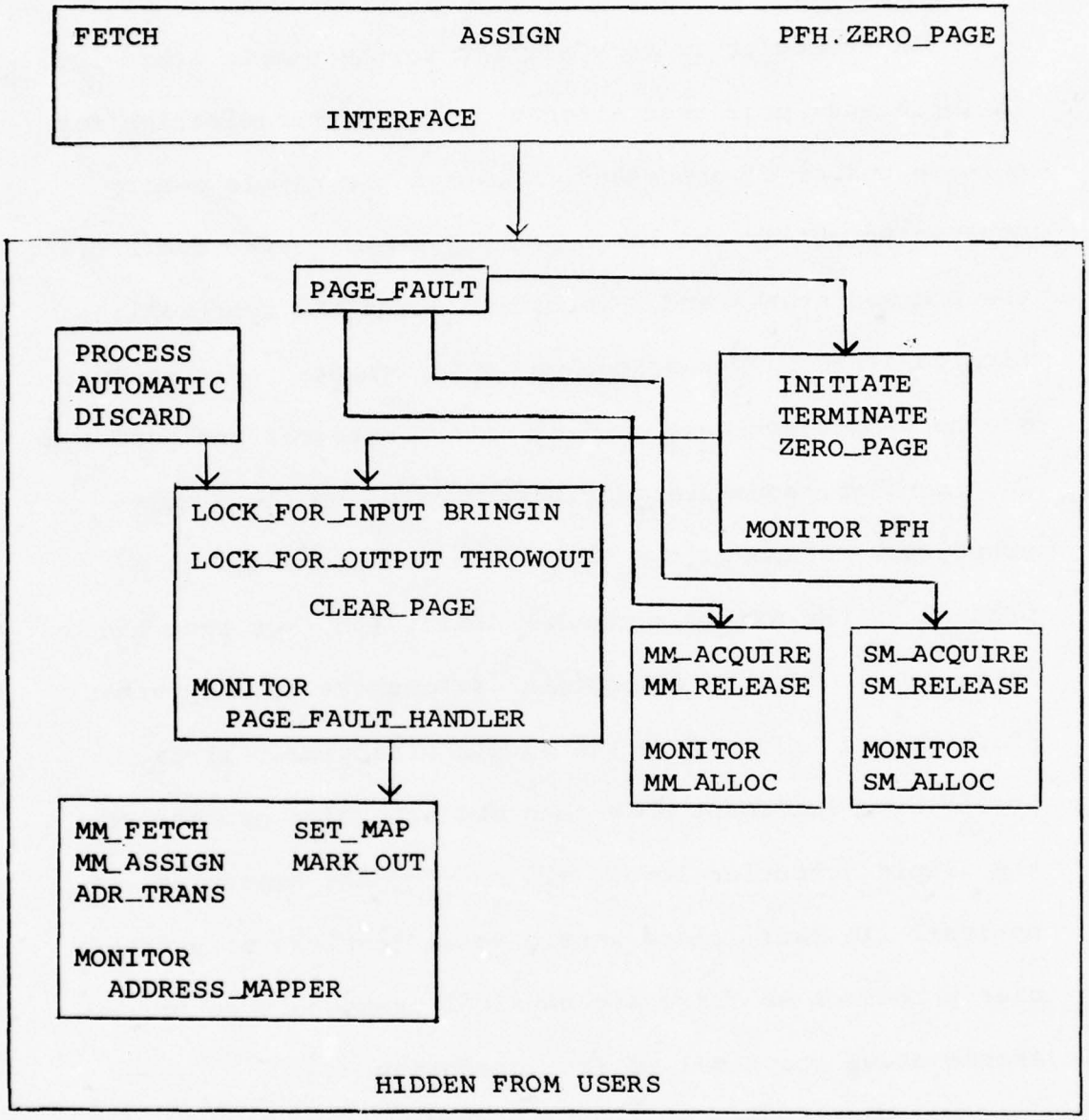


Figure 3.13

Structure of the Simple Memory Manager

3.5 SCHEDULER

The scheduler is very similar to the simple scheduler. It implements processor allocation and synchronization for a large number of processes. It uses the simple memory manager to obtain the large address space needed for storing the process states and information about the synchronization variables. The scheduler is implemented as a process at the simple scheduler level. Its procedures and data are in a monitor, scheduler_monitor, managed by the simple scheduler. Although the scheduler is implemented as a process at the simple scheduler level, the fact that the scheduler is in turn allocating processors to many other processes is hidden from the simple scheduler. It is possible to implement more than one scheduler process at the simple scheduler level, all sharing the scheduler_monitor. In fact, there should be at least as many scheduler processes as there are physical processors to be shared above the level of the scheduler.

The simple memory manager described in section 3.4 handles only those page-faults that occur in the scheduler processes; the page-faults suffered by the processes above the scheduler level are handled by the memory manager described in section 3.6. The hardware should be able to

distinguish between these two types of page-faults. The interrupts for the processes above the scheduler level can be handled by the scheduler in the same way as is done by the simple scheduler. Again, the hardware should be able to distinguish the two types of interrupt signals, one type to be serviced by the scheduler and the other type to be serviced by the simple scheduler. The hardware should provide the ability to mask interrupts and when a scheduler process is executing the interrupts for the memory manager should be masked.

A major distinction between the scheduler and the simple scheduler is the number of processes, monitors, and condition variables managed by them. The simple scheduler is intended to be used by a fixed and small number of processes. Further, these processes are assumed to be permanent. The scheduler, in contrast, must manage a variable and large number of processes and provide facilities for their dynamic creation and destruction. The monitors and condition variables at the simple scheduler level are also considered to exist for the life-time of the system, whereas the monitors and condition variables at the scheduler level may be created dynamically; thus, the scheduler must provide primitives for the dynamic

creation and destruction of monitors. These facilities were not needed at the simple scheduler level, because it was assumed that the processes, monitors, and condition variables will be created only once, at system generation time.

The capability to dynamically create and destroy processes introduces subtle problems. Although the processes are considered to be independent entities cooperating with each other, because of this cooperation the destruction of one process may affect another process non-deterministically. We can place the responsibility on the users of this level to ensure that when processes are destroyed they do not affect other processes non-deterministically. To ease the task of ensuring that processes are destroyed only under "safe" conditions, we will provide the following facility at this level: associated with each process is a lock-count, initialized to zero when the process is created; whenever a process enters a critical section of a program, one in which it is not safe to destroy the process, the lock-count associated with the process is incremented by one; when the process leaves the critical section the lock-count is decremented by one. A process can only be destroyed when its lock-count is zero. A process can destroy

another process only when the lock-count of the process to be destroyed is zero; if the process to be destroyed has a lock-count greater than zero, then the destroying process is suspended until the lock-count associated with the process to be destroyed decreases to zero; when the lock-count decreases to zero the process is destroyed and the destroying process is resumed. We will specify the action of destroying in two steps, first the process has to be stopped, a process can be stopped only if its lock-count is zero, and then it may be destroyed. When a process is destroyed its final state is returned to the destroying process. A process may not attempt to stop another process that is already stopped or that is in the midst of being stopped. The ability to destroy other processes may introduce deadlocks and the users of this facility must take care to avoid them.

A complication that arises in destroying processes is that the process to be destroyed may be in a blocked state, waiting on some condition, and when the process is destroyed it has to be removed from the list of processes waiting on that condition. To specify an efficient method of achieving this result we will need additional data structures and will have to modify some procedures. With the present

data structures, we will have to search through all lists of processes waiting on conditions to find the list containing the process to be removed. Instead, we will assume a procedure unblock, without specifying it in detail, that will be responsible for removing the process from the appropriate list.

Another complication that can occur in destroying processes is in the case of multi-processor systems. In a multi-processor system the process to be destroyed may be in the active state, that is, it may be executing on a processor. If the process to be destroyed (stopped) is active, we have to deactivate it by forcing an interrupt on the processor on which it is executing. We will assume that the procedure deactivate will accomplish this task. A process may be deactivated as follows: its status is changed to stopped; the procedure deactivate is invoked just before leaving the scheduler_monitor; procedure deactivate determines the identity of the processor on which the process is active and forces an interrupt on that processor; as a result of the interrupt the processor will cause the procedure dispatch to be invoked; when the procedure dispatch of the scheduler_monitor is entered (it is important that the procedure deactivate release

exclusion on the monitor scheduler_monitor after forcing the interrupt) it saves the state of the active process, and, since its state has been changed to stopped in the data structure of the monitor, it will not be considered for dispatching again.

The destruction of monitors also introduces problems. If a request to destroy a monitor occurs when a process is executing within the monitor or processes are waiting to enter the monitor or waiting on its condition variables, then the proper action to be taken is not clear. It should be the responsibility of the destroying process to ensure that such anomalous conditions do not occur. We will specify the procedure destroy_monitor to return an error indication when such an action is attempted.

The specification of the scheduler procedures and data structures are given in figure 3.14. The specifications of the procedures enter, exit, wait, signal, append and remove, and functions first and is_empty (and the relevant data declarations) are the same as the specifications in the simple scheduler level, except for the error checking; and are not repeated here. We have assumed that there can be at most N processes, M monitors, and C condition variables in a monitor that can exist at any time. The numbers N, M, and C

```
type monitorname = record in_use: Boolean; queue: 0..N end;
   condition = 0..N;

var process_set: set of 1..N;
    monitor_set: set of 1..M;
    no_of_conditions : array [1..M] of 1..C;
    process_state : array [1..N] of machine_state;
    m_name : array [1..M] of monitorname;
    c_name : array [1..M, 1..C] of condition;
    locked : array [1..N] of integer;
    status : array [1..N] of (active, ready, blocked,
                              stopped);
    stopper : array [1..N] of 1..N;

procedure create_ process (var p: 1..N; initial_state:
                           machine_state);
begin
   if process_set = empty then error_1
   else
      begin
         p := anyoneof(process_set);
         process_set := process_set - [p];
         process_state[p] := initial_state;
         status[p] := ready;
         locked[p] := 0;
         stopper[p] := 0;
      end;
end create_process;

procedure destroy_process (var final_state: machine_state;
                           p: 1..N);
begin
   if p in process_set then error_1
   else
      if (status[p] ≠ stopped) then error_2
      else
         begin
            final_state := process_state[p];
            process_set := process_set + [p];
         end;
end destroy_process;
```

Figure 3.14

Scheduler Specifications

```
procedure lock_process (p: 1..N);
begin
   locked[p] := locked[p] + 1; {Note a process can only
                                lock itself}
end lock_process;

procedure unlock_process (p: 1..N);
begin
   if locked[p] = 0 then error
    else
      begin
         locked[p] = locked[p] - 1;
         if locked[p] = 0  ∧  stopper[p] > 0  then
            begin
               status[p] := stopped;
               status[stopper[p]] := ready;
               dispatch(p);
            end;
      end;
end unlock_process;
```

Figure 3.14 (continued)

Scheduler Specifications

```
procedure stop_process (p1, p2 : 1..N);
   {p1 is the process desiring to stop process p2.}
begin
   if (p2 in process_set ∨ status[p2] = stopped) then
       error_1
   else
      if stopper[p2]>0 then error_2 {another process
         stopping p2}
      else
         if locked[p2] = 0 then
            begin
               case status[p2] of
                  active: begin
                             status[p2] := stopped;
                             deactivate(p2);
                          end;
                  ready: status[p2] := stopped;
                  blocked: begin
                              status[p2] := stopped;
                              unblock(p2);
                           end;
                  stopped: {error};
               end;
            end;
         else
            begin
               status[p1] := blocked;
               stopper[p2] := p1;
               dispatch(p1);
            end;
end stop_process;
```

Figure 3.14 (continued)

Scheduler Specifications

```
procedure create_monitor (var m: 1..M; ncv: 1..C);
var i: 1..C;
begin
   if monitor_set = empty then error
      else
         begin
            m := anyoneof(monitor_set);
            monitor_set := monitor_set - [m];
            no_of_conditions[m] := ncv;
            m_name[m].in_use := false;
            m_name[m].queue := 0;
            for i := 1 to ncv do
               c_name[m,i] := 0;
         end;
end create_monitor;
```

Figure 3.14 (continued)

Scheduler Specifications

```
procedure destroy_monitor (m: 1..M);
var i: 1..C;
    empty-q: Boolean;
begin
   if m in monitor_set then error_1
   else
      if (m_name[n].in_use ∨ ¬(is_empty(m.queue))) then
          error_2
      else
         begin
            i := 1;
            empty_q := true;
            while (i ≤ no_of_conditions[m] ∧ empty_q) do
               if c_name[m,i]>0 then empty_q := false;
            if ¬empty_q then error_3
            else
               monitor_set := monitor_set - [m];
         end;
end destroy monitor;

{initially}
begin
process_set := [1..N];
monitor_set := [1..M];
end;
```

Figure 3.14 (continued)

Scheduler Specifications

can be very large. The memory space required by the scheduler can be very large, but note that the simple memory manager allocates real resources (main memory and secondary memory) only when actually needed. Figure 3.15 gives the structure.

3.6 MEMORY MANAGER

The memory manager is very similar to the simple memory manager. The only difference between the simple memory manager and the memory manager is that the latter implements virtual address space for a large number of processes, whereas the former implements virtual address space for a small number of processes. In the specification of the simple memory manager, an implicit assumption was made that sufficient main memory space is available for the temporary space required in handling page-faults. Since the number of processes that can suffer page-faults and invoke the simple memory manager is small, the maximum temporary space required can be determined and reserved at the system initialization time. We cannot make the same assumption for the memory manager as the number of processes that can invoke it, and hence the maximum temporary space required, is very large. A certain amount of main memory space is reserved for temporary requirements and

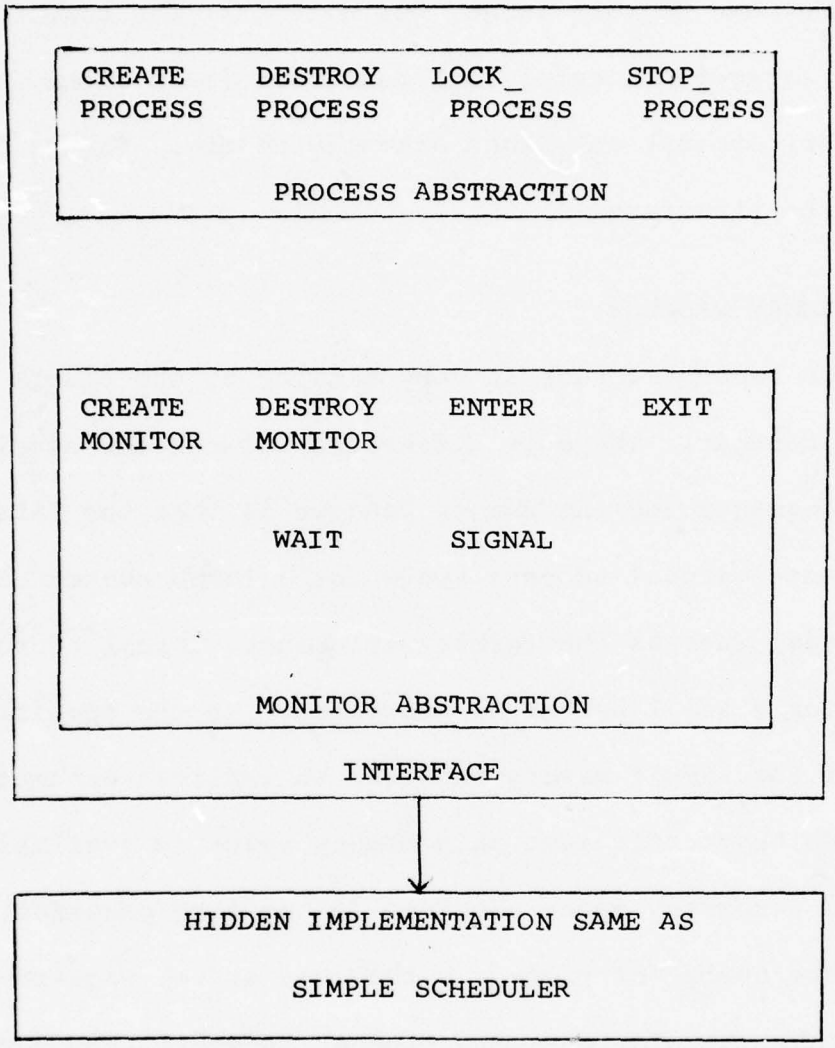


Figure 3.15

Structure of the Scheduler

the maximum number of page page-faults that can be handled concurrently in that space is determined. Whenever the number of page-faults to be handled increases beyond this limit, the processes invoking the page-fault procedure are delayed on a condition variable. Since the scheduler has a very large address space the number of processes waiting on the condition variable can be allowed to grow very large. The specification of the monitor and procedures needed for this solution are given below. The monitor page_fault_-limit is used to check if another page-fault can be handled. Note that as in the case of the simple memory manager we may need to organize the data structure of the memory manager itself into pages and may need multiple levels of memory management. The number of these levels is known at system initialization time and when we determine the limit on the number of concurrent page-faults to be handled we should consider that in the worst case, to handle each of these page-faults we may need to invoke all the levels of the memory manager. The monitor page_fault_limit is not used to check the limit for page-faults on the virtual map; the temporary space for handling these page-faults is reserved and guaranteed. We will divide the virtual address space managed by the memory manager into two parts. We

will use the variable v_min to denote the smallest virtual page number for the virtual address space of the user programs. The virtual page numbers from zero to v_min will be used for the virtual_map, the data structure of the memory manager. Note that it is not necessary to duplicate the monitor address_mapper since its data is always in the main memory and we can modify the procedure page_fault to determine which level of the memory manager to invoke. The various levels need only have different monitors pfh, page_fault_handler and the process automatic discard. The procedures for input, output and the monitors mm_alloc and sm_alloc can be shared. The main memory allocated for the use of memory manager should have at least twice as many page frames as the maximum number of concurrent page-faults allowed. The modified procedure page_fault and the monitor page_fault_limit are given in figure 3.16. We are using the function index(v: vpf) to determine the level of the appropriate monitor to be invoked, the monitors being declared as an array. The structure is given in figure 3.17.

3.7. CONCLUSION

The purpose of this chapter was to give the specification of the four levels of the solution to the many-process design problem and to demonstrate the use of

```
monitor page_fault_limit;
begin
var cpf: integer; {number of concurrent page-faults.}
    mpf: integer; {maximum number of concurrent page-faults.}
    pf_limit: condition;

procedure initiate;
begin
   cpf := cpf + 1;
   if cpf≥mpf then pf_limit.wait;
end initiate;

procedure terminate;
begin
   cpf := cpf - 1;
   pf_limit.signal;
end terminate;

 initialize.
   cpf := 0;
end page_fault_limit;

procedure page_fault (v: vpf);
var m:mmpf; s: sspf; p_out: Boolean;
begin
   if v ≥ v_min then page_fault_limit.initiate;
   pfh[index(v)].initiate(v,m,s,p_out);
   if p_out then
      begin
         input(m,s);
         pfh[index(v)].terminate(v,m,s);
      end;
   if v ≥ v_min then page_fault_limit.terminate;
end page_fault;
```

Figure 3.16

Memory Manager Specification

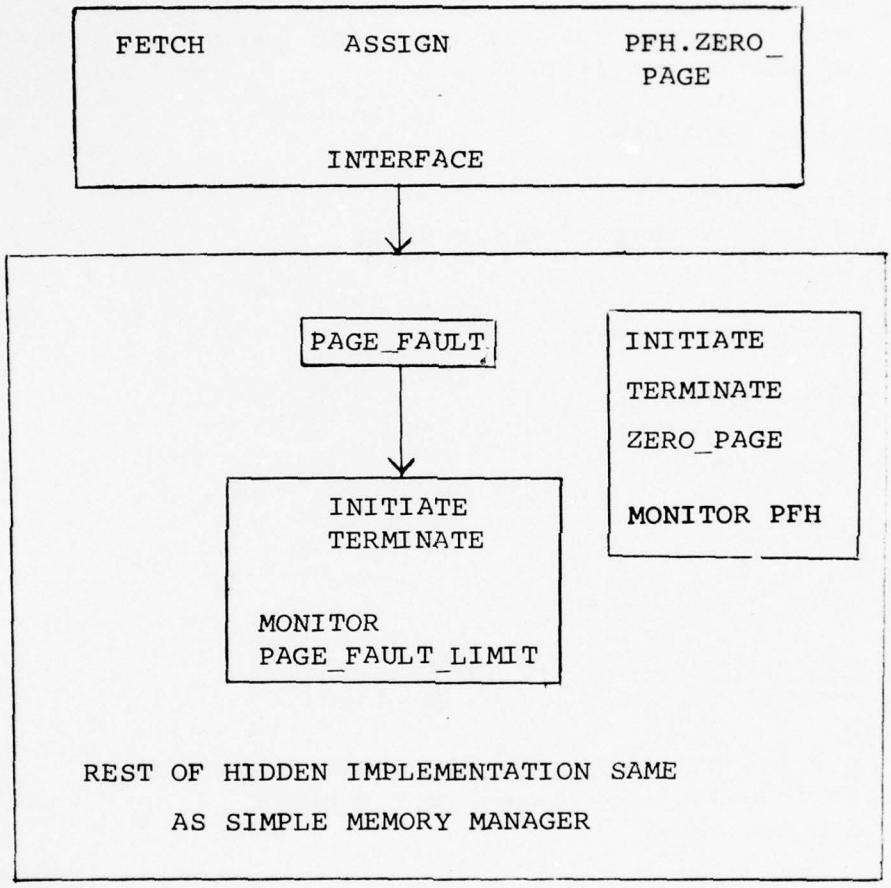


Figure 3.17

Structure of the Memory Manager

structured programming in specifying the solution. The concept of structured programming was explained in section 3.1 and its use is evident in the specifications. In the specification of the simple memory manager, section 3.5, we have shown the use of the technique of transformations mentioned in the article by Knuth [Knuth 1975] as a means of improving the efficiency of programs. It is our view, and the specification of the simple memory manager provides convincing evidence, that the use of transformations to modify a structured program which, although simple, has efficiency drawbacks into another program that is more efficient and also more difficult to understand is of great aid in understanding the efficient program and its verification. The technique can also be used for formally verifying programs as follows: the verification of the complex and more efficient program is done in two steps; the simpler program is verified and then the transformation from the simpler to the more complex program is verified. We have used this technique by verifying one specification of the simple memory manager in appendix C and then giving a verbal and less formal justification of the transformations to the final version of the specification.

Another use of structured specification is evident in our specification of the scheduler and the memory manager. The specification of the scheduler is obtained from the specification of the simple scheduler by extending it to include the case of dynamic creation and destruction of processes and monitors. This extension was simple to specify because of our use of procedure and data abstractions. The specification of the memory manager is a simple extension of the specification of the simple memory manager to handle the problem of limited temporary storage space. It is our conclusion that structured programming techniques are not only applicable but also very useful in the specification of operating system programs. Further the use of PASCAL in specifying the programs shows that it is possible to give detailed specification of operating system programs in a high-level, and a machine-independent, language. Figure 3.18 presents the structure of the four-level solution and the facilities they provide.

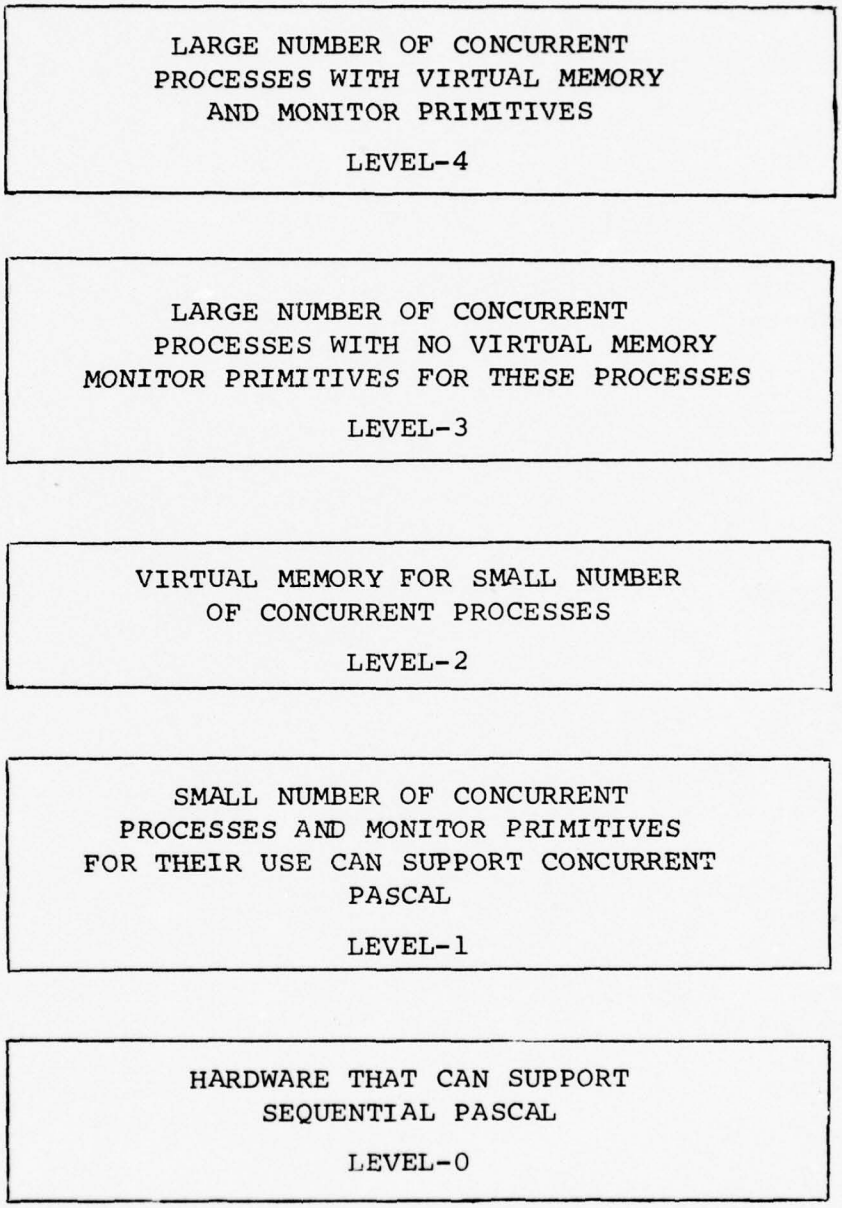


Figure 3.18

Structure of the Four-level Solution

CHAPTER 4. VERIFICATION

This chapter explains the concepts and methodology used for verifying the operating system specification and the properties verified. In section 4.1 the methodology used is explained in the context of sequential programs and in section 4.2 the new concepts and extensions of the methodology for concurrent programs is explained. Section 4.3 deals with the properties of the simple scheduler level and section 4.4, with the properties of the simple memory manager.

4.1. METHODOLOGY FOR VERIFICATION

The methodology to be used for verifying the programs is known as the verification rules technique [Manna '74] and was proposed by Hoare [Hoare 1969]. This technique is based on the inductive assertion approach of Floyd [Floyd 1967]. The technique proposed by Hoare was for sequential programs.

Hoare introduced the notation

$$\{P\}B\{R\}$$

to be interpreted as: "If the assertion P is true before

initiation of a program B, then the assertion R will be true on its completion."

Hoare then introduced a deductive system, which consists of an assignment axiom and rules of inference for composing the proofs for large program segments from the proofs of smaller program segments.

The programming language PASCAL [Wirth 72] has been axiomatized [Hoare and Wirth 72] and follows Hoare's deductive system. Since our programs are written in PASCAL, we will be using these axioms and rules [given in the appendix A]. It should be noted that these axioms and rules of inference apply only for sequential programs. The extensions to concurrent programs are explained in the next section.

Hoare's verification rules only prove partial correctness. That is, the axioms and rules of inference give no basis for a proof that the program successfully terminates. In our proofs we will give a separate proof of termination. In case of sequential programs we will prove the absence of infinite loops and in case of concurrent programs, we will also give a proof of absence of deadlocks.

4.2. VERIFICATION OF CONCURRENT PROGRAMS

In verification of concurrent programs, an important

notion is that of a process. A process, for our purposes, may be viewed as a locus of control, which moves among the programs as they are executed on a processor. There is in general no one to one relationship between programs and processes. In concurrent programs, the same program may be executed by different processes at the same time and/or at different times. In sequential programs, one can associate a program with a process and verifying a program will imply that the results will hold for the process as well. This is not true in case of concurrent programs. There are two main issues to be dealt with, the exclusive access of shared variables, and the execution of a program by more than one process, i.e. it is possible for one process to execute a part of a program, and another process to execute the rest. We will deal with these issues separately. First we will consider the execution of a single program segment by more than one process and then the problem of exclusive access to shared variables.

The point to be noted in this section is the distinction between the verification of a program and the verification of a program's execution on behalf of a process, or in short process verification. Verification of a

program, or at least a sequential program, consists of verifying that if the input assertion is satisfied at the program's initiation, then on termination of the program, the output assertion is satisfied. The distinction between program and process verification arises in case of concurrent programs, because when a program terminates, the process which initiated the execution of the program may not be the same as the process which terminates the execution. Hence the output assertion at the end of program termination is not usually the same as the output assertion desired by the process initiating the execution of the program. So we define the concept of a process verification as follows.

{Pprocess}B{Rprocess} is to be interpreted as: "if the assertion Pprocess is satisfied when the process initiates the execution of B (i.e. the process is in the active state), then on termination of the program B and on the state of the process being active the assertion Rprocess will be satisfied."

An example of the use of these concepts appears in the proof of the simple scheduler level programs and is explained in Section 4.3. We should note that the distinction between a process verification and a program

verification need be made only for programs of levels 1 and 3 in our case and that at other levels, (i.e. levels 2 and 4), they are equivalent. This distinction in general has to be made for programs which implement processor sharing among concurrent processes.

Notion of exclusive access.

In sequential programs, the concatenation axiom holds. In concurrent programs, with shared variables, the concatenation axiom stated is not valid. This is so because processes may modify the value of shared variables. Now in extended PASCAL, the shared variables are restricted to be within monitors. Thus we need to extend the axioms. A monitor is usually responsible for a shared resource. It would be very convenient if all operations on a resource were to be performed by the monitor on behalf of the processes, but it is sometimes convenient to have a monitor simply as a resource allocator. If a resource is a shared resource then we may need the assurance that a resource is exclusively owned before we can assert its properties (value). Thus there is an implicit assumption that all resources are exclusively owned. (This assertion is necessarily valid in sequential programs). Initially a monitor is the sole possessor of its resources. Some of

the monitor procedures can be used to acquire the resources. It should be proved that when a monitor allocates a resource to a process it cannot reallocate the resource without reacquiring it first. Thus the process which acquired the resource can claim exclusive access if and only if it can release access to the resource. The acquiring process may deposit the resource with another monitor.

To check for validity of exclusive access, we will associate an invariant with each resource, i.e. a monitor or a process possesses a resource if and only if the invariant is satisfied. Thus if a monitor owns a resource before a process exits from it but not immediately after, we can deduce that exclusive access to the resource has been passed to the process. In general we may distinguish between types of access to a resource and associate invariants with particular types of access to a resource.

The explicit statement of exclusive access in the assertions helps to check that the synchronization conditions are satisfied.

Assertions using past and future states of shared data

Another difference between sequential and concurrent program is the nondeterministic state of the shared data that is operated on by concurrent programs. When operations

are performed on shared data, using monitor procedures, all that we can assert is an invariant on initiating the execution of the monitor procedure and on terminating its execution. We cannot make any assertions on the state of the monitor data before we enter the monitor or after we exit from it. When we invoke a monitor procedure we wish to affect the monitor data in a particular fashion. The desired effect is specified by the monitor state at the initiation and termination of the procedure. The invariants are usually insufficient for this purpose. What we need is some means of stating the state of the monitor data at the instant we leave the monitor assuming its state at the instant we enter the monitor. We have used free variables to indicate the state of the monitor data on entry, with the assumption that the assertion on the monitor data refers to the actual instant when monitor entry is gained and not the instant when monitor entry is attempted. To indicate the state of the monitor data on exiting from the monitor we have used predicates that assert existence of states of the monitor data on exit from the monitor in terms of the state on entry to the monitor. The use of this approach is explained in appendix C.

Our approach is different from that of Owicki and

Gries [Owicki and Gries 1975] who use auxiliary variables and that of Howard [Howard 1975] who uses history variables to prove termination. The use of auxiliary variables and history variables requires proving the monitors with the addition of these variables to the monitor data and operations on them, and deducing the proofs of the original monitor from the proofs of the monitors augmented with the operations on auxiliary variables.

The use of history variables facilitates proving assertions about fair scheduling and properties such as queueing disciplines. We needed predicates on past states of the monitor, because we had independent monitors that interacted with each other and in the otherformalisms of proving concurrent programs [Hoare 1974, Howard 1975, Owicki and Gries 1975] such problems were not handled.

The use of auxiliary variables [Owicki and Gries 1975] to prove assertions regarding a system of parallel processes can be very useful. Again, the examples dealt with were more restrictive than our case because they considered parallel processes within cobegin ... coend statements, and only single shared programs. In our case, we do not know the number of parallel processes or their environments when they use the monitors, nor do we restrict our parallel

programs from using other independent parallel programs.

It is possible to extend the auxiliary variable approach to the case of interdependent monitors. Unlike Owicki and Gries who make auxiliary variables resources of the monitor (conditional critical region) what we need to do is to pass these as auxiliary parameters, that will return information on the state that existed in the monitor. Thus, associated with each of the predicates we have used in our proofs, we could have created auxiliary variables that would have conveyed exactly the same information as the predicates.

Absence of deadlocks

A problem in concurrent programs which is not present in sequential programs is that of deadlocks. Deadlock is a situation in which two or more processes are unable to make any progress because of conflicting resource requirements [Coffman et al. 1971]. A related concept is that of permanent blocking. A process is said to be permanently blocked if the resource allocation strategy is such that a process is denied a requested resource for an infinite time [Holt 1971].

The necessary conditions for the existence of a state of deadlock are [Coffman et al. 1971]:

1) Processes claim exclusive control of resources they require ("mutual exclusion" condition).

2) Processes hold resources already allocated to them while waiting for additional resources "wait for" condition).

3) Resources cannot be taken away from the processes holding them until they are released ("no preemption" condition).

4) A circular chain of processes exists, such that each process holds one or more resources that are being requested by the next task in the chain ("circular wait" condition).

It is also known [Coffman et al. 1971] that the "circular wait" condition is a necessary and sufficient condition for a deadlock, assuming the other three conditions given above are satisfied. In the case of resources controlled by monitors, the first three conditions are satisfied and the unsatisfiability of the circular wait condition is sufficient to assume absence of deadlocks. Further, in the case of monitors, a process is waiting on at most one resource (condition variable) and under these circumstances a FIFO resource allocation strategy will prevent permanent blocking.

In this section we wish to avoid deadlock within a monitor. We will develop sufficient conditions, so that a process is never waiting on a condition variable inside a monitor forever. Note that with fair scheduling and assuming termination of monitor programs, it is impossible for a process to be denied access to a monitor forever.

To develop sufficient conditions to avoid deadlocks in monitors, it is helpful to define the following terms.

Consider a system of monitors and concurrent processes. We define a hierarchy of monitors as follows. Associate a level with each monitor, such that if no procedure in the monitor calls any procedure of any other monitor then its level is 0. No procedure of a monitor may call any procedure of another monitor which has a lower level. A procedure A is said to call another procedure B, if in A's definition there is a procedure statement for the procedure B. A monitor's level is the lowest integer consistent with the above constraints.

If such a hierarchy cannot be determined for a system of monitors, then there is a potential deadlock. (This is so because it would imply a loop, and it is impossible to enter a monitor for the second time without releasing access to it first).

Associated with each "wait" statement in a procedure we define a wait-p-path as a sequence of procedures, such that the first procedure in the sequence contains a call on the procedure with the wait statement and the i-th procedure in the sequence contains a call on the (i-1)_th procedure.

Corresponding to each wait-p-path, we define the wait-m-path as the sequence of monitors that contains the corresponding procedures in the wait-p-path, such that no monitor name is repeated, i.e. if two procedures belonging to the same monitor appear (they have to be consecutive) then the monitor name appears only once.

Associated with each "signal" statement in a procedure we define a signal-p-path as a sequence of procedures, such that the first procedure in the sequence contains a call on the procedure with the signal statement and the i_th procedure in the sequence contains a call on the (i-1)_th procedure.

Corresponding to each signal-p-path, we define the signal-m-path as the sequence of monitors which contain the corresponding procedures in the signal-p-path. Such that no monitor name is repeated, i.e. if two procedures belonging to the same monitor appear (they have to be consecutive)

then the monitor name appears only once.

Condition N1:

For every wait-p-path, for a condition variable, there exists a signal-p-path for the same condition variable such that (a) no two procedures in the path belong to the same monitor, i.e. the corresponding wait-m-path and signal-m-path are disjoint, and (b) any procedure that is common to the two paths is such that the call on the succeeding procedure of the signal-p-path occurs before the call on the succeeding procedure of the wait-p-path.

Theorem 4.2a:

The condition N1 is a necessary condition for the absence of deadlocks.

Proof:

The proof is by contradiction. Assume that condition N1(a) and N1(b) are not satisfied. If condition N1(a) is not satisfied then there exists a wait-p-path such that for the corresponding wait-m-path, there is no disjoint signal-m-path. In this case, if the wait statement is executed on a condition variable following the wait-p-path there would be no way to execute a signal statement on the same condition variable, since all paths to the signal

statement will be blocked at the common monitor. (Note that when a wait statement is executed in a monitor exclusion on enclosing monitors is not released.) If condition N1(b) is not satisfied then it implies that there exists a wait-p-path such that it has a common procedure with every signal-p-path and that the call on the succeeding procedure of the wait-p-path occurs before the call on the succeeding procedure of the signal-p-path. In this case all the signal-p-paths will be blocked on a common procedure, since the procedures will be executed sequentially in the path.

We will now give sufficient conditions, so that the procedures in the monitor themselves are not such as to cause deadlock. The sufficient conditions are given in the order of their restrictiveness, that is we will first give a sufficient condition, which if satisfied will guarantee that there is no inherent deadlock, but which may not be satisfied in many monitors, which also have no inherent deadlocks. We will then make this condition less restrictive.

We will make the following assumptions for all the conditions.

Associated with each condition variable c_i is a

boolean condition C_i, such that:

$\{I \wedge C_i\}\ c_i.\text{signal}\ \{I\}$ and $\{I \wedge \neg C_i\}\ c_i.\text{wait}\ \{I \wedge C_i\}$

where I is the invariant associated with the monitor. Further associated with each procedure p_i of the monitor is an input assertion P_i, which is to be satisfied on entry to the procedure p_i. We are also assuming that provided there is no inherent deadlock, all procedures terminate.

Condition - S1:

For every condition variable c_i, there should exist at least one procedure p_i, such that p_i has no wait statements and contains a signal statement on the condition variable c_i and $I \wedge \neg C_i \Rightarrow P_i$.

Proof:

The condition guarantees, that whenever a process may be waiting on a condition variable, there always exists an executable path to a corresponding signal statement in the monitor.

Condition - S2:

For every condition variable c_i, let X_i be the set of procedures which contain a corresponding signal statement, and let $Y_{i,j}$ be the set of condition variables in the path of the signal statement in the procedure p_j, $p_j \in X_i$ then

$$\bigvee_{p_j \epsilon X_i} (I \wedge \neg C_i \Rightarrow P_j \bigwedge_{c \epsilon Y_{i,j}} (C))$$

Proof:

The implication assures that a process attempting to signal c_i, will not be itself blocked on any condition variable in its path.

Condition S3:

For every condition variable c_i let $S(c_i)$ be the logical expression, such that if it evaluates to true then a signal operation on the same condition variable can be executed. Let X_i be the set of all procedures p_j such that a signal statement on the condition variable c_i appears in it. (We are assuming that there is at most only one signal statement per condition variable in a procedure. If for some reason there is more than one, we will consider only the first. Let $Y_{i,j}$ be the set of condition variables with wait operations which appear in the procedure p_j, in the set X_i. Then, $S(c_i)$ may be defined as:

$$S(c_i) = I \wedge \neg C_i \Rightarrow \bigvee_{p_j \epsilon X_i} (P_j \bigwedge_{c_k \epsilon Y_{i,j}} (C_k \vee S(c_k)))$$

Proof:

The proof that $S(c_i)$ is sufficient for a signal to be

operated on c_i is obvious, since it guarantees that there exists at least one path free to a signal operation. Note we may find occurrences of $S(c_i)$ on the right hand side on simplification, after obtaining a disjunctive normal form, all disjuncts containing it may be removed. (They evaluate to false).

Given that there exists a disjoint path through the nested monitors to a procedure with a signal operation, which can be executed, it is still necessary that there exist in the system a process, that is guaranteed to take that path. This can only be shown, if we know the states of the system and the paths a process may take and it depends on the process definition. Thus conditional on there existing a process, that will take the desired path, the sufficient condition for a process not to wait on a condition variable forever may be stated as follows:

Let Z_i be the set of procedures in X_i, such that $p_j \in Z_i$ implies that $S(c_i) = p_j \bigwedge_{c_k \in Y_{i,j}} (C_k \vee S(c_k))$.

Then if for every wait statement on c_i and the corresponding wait-p-path there exists a signal-p-path satisfying N1, to p_j such that $p_j \in Z_i$ then a process executing the wait instruction will (not wait forever) eventually resume

execution. Note that we can modify the condition S3, as follows:

$$S(c_i) = I \wedge \neg C_i \wedge \neg \text{is_empty}(c_i.\text{queue}) \Rightarrow \bigvee_{p_j \in X_i} (P_j \bigwedge_{c_k \in Y_{i,j}} (C_k \; S(c_k)))$$

Since we are only interested in the condition being satisfied if there is any process waiting on the condition variable and is-empty(c_i.queue) is a boolean function which evaluates to true if no process is waiting on the condition variable c_i.

The above conditions are sufficient only if we assume that a process never waits on a condition variable c when C is satisfied. To prove this assumption to be valid we need stronger invariants and modified assertions for the wait and signal operations. We have to include in the invariant I the assertion $\forall c$: C => c.queue = empty where c.queue is the list of processes waiting on the condition variable c.

For partial (that is logical correctness, not including proof of absence of deadlocks) verification the proof rules associated with the wait and signal operation are:

$$I\{c.\text{wait}\}\ I \wedge C$$

$$I \wedge C\ \{c.\text{signal}\}\ I$$

where I does not include the relation C => (c.queue = empty). For proving absence of deadlocks we need stronger assertions.

$$I \wedge \neg C \wedge I_d \ \{c.wait\} \ I \wedge C \wedge D_c$$

$$I \wedge C \wedge D_c \ \{c.signal\} \ I \wedge I_d$$

where I_d is the deadlock invariance

$$\forall c: C \Rightarrow (c.queue = empty)$$

and D_c is the deadlock condition associated with the condition variable c. Normally D_c will imply I_d for all condition variables except c (it may not satisfy I_d for the condition c, if the signal operation is executed when c.queue is greater than one because in that state C will be true but c.queue will be nonempty). Further, D_c may also imply a relation other than C, that is required for proving I_d on termination of the procedure under the condition that the wait operation was executed.

An inspection of the assertions shows that if the wait operation was not executed, because C was already satisfied then the assertion after the <u>if</u> clause containing the wait statement (Note we <u>never</u> execute a wait operation on a condition c if C is true; if we do then the deadlock-free property is not proved by our method, in fact it will

violate the relation I_d for the condition variable c) will be $I \wedge C \wedge I_d$, whereas if the wait operation was executed then it may be $I \wedge C \wedge D_c$ and these may be contradictory, (that is $I_d \wedge D_c$ = false). We have to prove the absence of deadlocks under both of these mutually exclusive conditions. As noted earlier, it is obvious that the invariant relation I_d is satisfied on exit from the procedure if the wait operation was never executed. So we need only verify the procedures assuming that the wait instruction will be executed (Note: if there is more than one wait instruction in a procedure, we have to verify all paths that contain at least one wait instruction).

Also observe that when the signal operation is executed and there is no process waiting on the condition variable, then the operation has no effect and

$$I \wedge C \wedge D_c \ \{c.signal\} \ I \wedge I_d$$

may not be satisfied. Clearly when there is no process waiting on the condition we need an assertion of the form

$$I \wedge I_d \wedge C \ \{c.signal\} \ I \wedge I_d$$

we could of course satisfy both the assertions by choosing $D_c' = (c.queue = empty \Rightarrow I_d) \wedge (\neg(c.queue = empty) \Rightarrow D_c)$ and using D_c' instead of D_c. If we use D_c' in verification

we have to consider the two cases separately anyway, instead the approach we are adopting is to specify D_c and verify for the case when c.queue $\neq$ empty.

We can prove the absence of deadlocks considering only the invariant relation I_d since $I \wedge I_d \Rightarrow I_d$ and $I \wedge I_d \Rightarrow I$, and the partial verification of the procedures with the invariant I is still valid.

Howard [Howard 1975] has given proof rules to ensure that a process that can legally continue is not blocked. However, his proof rules are too strong, i.e. many useful monitors may not satisfy his proof rules. He demands as an invariant an assertion E such that E implies the negation of all conditions associated with the condition variables of the monitor. Thus it would not even be possible for two mutually exclusive conditions to exist in the monitor such as "buffer is full" and "buffer is empty" in a producer-consumer problem.

4.3. VERIFICATION OF THE SIMPLE SCHEDULER

In this section the properties of the simple scheduler that are relevant to the users of the level are proved. In proving these properties, we will make use of the properties of the simple scheduler program that have been

verified in the appendix B.

The properties to be proved about monitor access are (assuming proper initialization, i.e. m.in-use=false, for the monitor m):

P1) No more than one process ever has concurrent access to the monitor, at any time.

P2) No process is denied access to the monitor forever, assuming that a process which has access to a monitor eventually releases the access.

The properties to be proved about the condition variables, are:

P3) When a process executes a "wait" operation on a condition variable, it is resumed only after a subsequent signal operation on the same condition variable, and when a process is so resumed, no other process gains access to the monitor after the signal (i.e. whatever is the state of the monitor before the signal, the same state will be found by the resumed process after the wait).

P4) When a process executes a "signal" operation on a condition variable it is blocked only if there is a waiting process on the condition variable and when blocked is resumed only when no process is in the

monitor.

Property P1 may be proved by showing the validity of the invariant:

(m.in_use = false ≡ no-process in the monitor) ∧

(m.in_use = true ≡ exactly-one-process in the monitor)

This is clearly true after initialization and is maintained by the four procedures - enter, exit, wait and signal, (this is proved in appendix B).

Thus there can never be more than one process in the monitor. Property P2 is proved as follows:

Whenever a process is waiting to enter the monitor m, m.queue is not empty. We show that m.in_use = false∧ m.queue ≠ empty is not possible, i.e. m.in_use = false => m.queue = empty.

This is clearly true at initialization and also after each of the four procedures. (The proof is in the appendix B).

Property P3 is proved as follows:

After a wait operation on a condition variable by process p, it is appended to a queue on the condition variable and blocked which implies it cannot execute any other statement nor join any other queue. The only way a process leaves the queue and regains active

status is by a signal operation on the queue. The access to the monitor passes directly from the signalling process to the waiting process and thus the state of the monitor variables cannot be changed. The resumed process starts execution just after the wait statement.

The proof of property P4 follows from the proofs of the procedures signal, wait and exit in appendix B.

Thus by associating an invariant I with a monitor, which is true after initialization and on exit from the monitor as well as a precondition for all wait operations, and associating a boolean condition C, with a condition variable c which is implied by all preconditions of signal operations on c, we can assert the following:

$\{process\text{-}in\ (m) \neq p\}$ enter (m,p) $\{process\text{-}in\ (m) = p \wedge I\}$

$\{process\text{-}in\ (m) = p\}$ exit (m,p) $\{process\text{-}in\ (m) \neq p\}$

$\{process\text{-}in\ (m) = p \wedge I\}$ wait(m,c,p) $\{process\text{-}in\ (m) = P \wedge C\}$

$\{process\text{-}in\ (m) = p \wedge C\}$ signal (m,c,p) $\{process\text{-}in(m) = p \wedge I\}$

The proofs of these follow easily from the proof of the properties P1, P2, P3 and P4 and are given in the appendix B. "process-in (m)" is the identity of the process inside the monitor m.

4.4. VERIFICATION OF THE SIMPLE MEMORY MANAGER

What we would like to prove is that assign (va,w) changes the value of vm[va] to w and on subsequent fetches without intervening assigns and calls on clear_page we would get back the same value i.e.

{} assign (va,w0);s1;s2;s3;fetch (va,w) {w = w0}

provided s1,s2,s3 ≠ assign (va,x) ∨ clear_page (va.vpno).

The proof follows from the proof of the procedures clear_page and the procedures assign and fetch, since they operate using the procedures mm_assign and mm_fetch of the monitor address-mapper, and all other procedures keep the value of vm invariant. This is shown by considering the mapping vm onto mm, sm, and the all-zero contents, and by the proof of the invariants of the monitors.

The other property we would like to prove is the termination of the procedures assign and fetch. To prove termination we have to make certain assumptions on the behaviour of the system, which will result in the probability of termination tending to one as time tends to infinity. (The proof is given in the appendix C).

We have to make use of statistical arguments to prove termination, because there is no guarantee that once a page is brought into the main memory, it will not be

thrown out before it is accessed by a process. We could devise a solution in which a page is not thrown out until at least one reference is made to the page. This scheme has several drawbacks. The first is that it can cause the main memory to be filled with useless pages which cannot be thrown out, if a process suffering a page-fault can be aborted after the page-fault is handled but before an attempt at referencing the page is made. Another drawback is that it does not completely solve the problem by itself. For if more than one process is waiting for the page, then it is difficult to guarantee that all processes make at least one reference to the page, before it is thrown out. Finally it is not clear whether the advantage of ensuring that the page is not thrown out till all processes waiting for it have referenced it outweigh the overhead of such schemes, especially, when we can prove termination by statistical arguments and ensure a low probability of consecutive page-faults on the same reference by other means.

Another property we may like to prove is that the process automatic_discarder discards pages cyclically, in the simple scheme verified in the appendix C.

Note that the proofs in the appendix are given for the simplified version of the Simple Memory Manager. The

transformation made to it for increase of efficiency has been shown to preserve the logical correctness.

We also demonstrate that there are no deadlocks in the memory manager, that is every process which is waiting on a condition will be eventually resumed. This is the proof of termination of the procedures assign and fetch.

CHAPTER 5 CONCLUSION

The purpose of this thesis was to obtain a solution to the many-process problem and to investigate the use of hierarchical levels of abstraction as a design methodology for operating systems; the use of structured programming techniques in the specification of the system; the feasibility of, and development of techniques for, verifying concurrent programs such as operating system programs.

We obtained a solution to the many-process problem (in Chapter 2) and have found that hierarchical levels of abstraction methodology simplifies the conception of the solution and helps avoid potential deadlocks in the system. In chapter 3 we presented a specification of the four levels of the system and have, we believe, demonstrated the usefulness of structured programming techniques for specifying operating system programs. The use of the step-wise refinement (transformation) as shown in section 3.4 is of great aid in understanding the final specification of the simple memory manager, which is quite complex and a reasonably large segment of the system. Earlier version of chapters 2 and 3 appeared in [Bredt and Saxena 1974] and [Saxena and Bredt

1975].

In chapter 4 we developed some techniques for verifying concurrent programs. The notion of process assertions as distinct from procedure assertions is important in understanding the verification (and use) of the monitor primitives. We have found the notion of exclusive access of much use not only in verifying partial correctness but also in verifying absence of deadlocks. In chapter 4, we also developed some sufficient conditions for verifying the absence of deadlocks partially, that is, when we do not know the behaviour of the processes that will use the system of monitors or in particular how they would use the monitors. This is useful in verifying that the system itself does not have inherent deadlocks. Since in our case, we restricted access to the monitors to be only through the re-entrant procedures fetch and assign and the monitor procedure clear_page, we were able to prove that the system is free of deadlock and that these procedures always terminate.

One of the limitations of the work reported here is that the system specified is not implemented. It would be useful to implement the system and measure its performance to find the bottlenecks in the system and to see if these can be removed by refining the solution, using

the hierarchical levels of abstraction approach and the structured programming techniques.

Another limitation is that we considered only the problem of processor and memory management. It would be useful to extend the solution to input/output and file management so that the system can be used by less sophisticated users.

A useful area of further research is in the development of a formalism for verifying concurrent programs. In our proofs, we had to make use of state dependent predicates that are of a different nature than the assertions for sequential programs in that they do not refer to current state but assert that a particular state existed in the past. It would be useful to develop a unified theory of verifying programs which would include the verification of sequential programs as a special case.

We hope that this work will encourage the use of levels of abstraction approach and structured programming techniques in the design and specification of operating systems. The fact that it is possible to verify operating system programs leads us to believe that proper design can result in understandable program systems, even if they have parallelism.

APPENDIX A

VERIFICATION RULES

In this appendix we will list the axioms and rules of inference used in the verification of the programs. The axioms for the data types of PASCAL are given in [Hoare and Wirth 1972] and are not repeated here. The notation and concepts as well as the axioms and rules of inference are those given in [Hoare and Wirth 1972] except for those dealing with monitors.

Concepts and Notations

The notation used are mainly those of symbolic logic. They are supplemented by the following conventions:

P, P_1, Q, R stand for propositional formulas.

S stands for program statements.

x,y stand for variable names (y not free in P or Q).

e stands for an expression.

B stands for a Boolean expression.

p stands for a procedure name.

{P}S{Q} -- where P and Q are propositional formulas of logic and S is a part of a program. Explanation: If

P is true of the program variables before executing the first statement of the program S, and if the program S terminates, then Q will be true of the program variables after execution of S is complete.

P_e^x -- where P is an expression or formula, X is a variable, and e is an expression. Explanation: The result of replacing all free occurrences of x in P by e. If e is not free for x in P, a preliminary systematic change of bound variables of P is assumed to be made.

$\frac{A,B}{C}$ -- where A, B, and C are propositional formulas. Explanation: A rule of inference which states that if A and B have been proved, then C may be deduced.

$\frac{A,B\quad C}{D}$ -- where A, B, C, and D are propositional formulas. Explanation: A rule of inference which permits deduction of D if A and C are proved; however it also permits B to be assumed as a hypothesis in the proof of C. The deduction of C from B is known as a subsidiary deduction.

It is assumed that, with the exception of program material, all letters stand for formulas of some suitably chosen logical system (usually enclosed in braces). The formulas of this system are presumed to include:

(a) all expressions of the programming language;

It is this property that is assumed in proving assertions about expressions containing calls of the function f, including those occuring within S itself and in other declarations in the same block. In addition, assertions generated by the parameter specifications in L may be used in proving assertions about S.

procedure p(L): S

Let $\underline{x}$ be the list of explicit parameters declared in L ; let $\underline{y}$ be the set of global variables occuring in S (implicit parameters), let $x_1 \ldots x_m$ be the parameters declared in L as variable parameters, and let $y_1 \ldots y_n$ be those global variables which are changed within S. Given the assertion $\{P\}S\{Q\}$, we may deduce the existence of functions f_i and g_j satisfying the following implication:

$$P \Rightarrow Q^{x_1 \quad \ldots \quad x_m \,, \quad y_1 \quad \ldots \quad y_n}_{(f_1(\underline{x},\underline{y}) \ \ldots \ f_m(\underline{x},\underline{y}),\ g_1(\underline{x},\underline{y}) \ \ldots \ g_n(\underline{x},\underline{y}))}$$

for all values of the variables involved in this statement.

It is this property that is assumed in proving assertions about calls of this procedure, including those occuring within S itself and in other declarations in the same block.

The functions f_i and g_j may be regarded as those which

map the initial values of $\underline{x}$ and $\underline{y}$ on entry to the procedure onto the final values of $x_1 \ldots x_m$ and $y_1 \ldots y_n$ on completion of the execution of S.

Simple statements

Assignment statements:

$$\{P_y^x\}x := y\{P\}$$

In the case where x is an indexed variable, we introduce the convention that

$$P_y^{a[i]} \quad \text{means} \quad P^a_{(a,i:y)},$$

where a,i:y is the array obtained from a by assigning y to the i_th element and leaving the other elements unchanged and if x is a field designator, we introduce the convention that

$$P_y^{r.s} \quad \text{means} \quad P^r_{(r,s:y)}.$$

Compound statements

$$\frac{\{P_{i-1}\}S_i\{P_i\}, \quad \text{for } i = 1 \ldots n}{\{P_o\}\underline{\text{begin}}\ S_1; S_2 \cdots S_n\ \underline{\text{end}}\{P_n\}}$$

If statements

$$\frac{\{Q_1\}S_1\{R\}, \{Q_2\}S_2\{R\}, \quad P \wedge B \supset Q_1, \quad P \wedge \neg B \supset Q_2}{\{P\}\ \underline{\text{if}}\ B\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\{R\}}$$

$$\frac{\{Q\}S\{R\},\ P\wedge B\Rightarrow Q,\ P\wedge\neg B\Rightarrow R}{\{P\}\ \underline{\text{if}}\ B\ \underline{\text{then}}\ S\{R\}}$$

Case statements

$$\frac{\{Q_i\}S_i\{R\},\ P\wedge(x=k_i)\Rightarrow Q_i,\ \text{for } i = 1..n}{\{P\}\ \underline{\text{case}}\ x\ \underline{\text{of}}\ k_1:S_1;k_2S_2;\ldots k_n:S_n\ \underline{\text{end}}\ \{R\}}$$

Note: $k_a, k_b \ldots k_n:S$ stands for $k_a:S;\ k_b:S;\ \ldots\ k_n:S$

While statements

$$\frac{\{Q\wedge B\}S\{Q\}}{\{Q\}\ \underline{\text{while}}\ B\ \underline{\text{do}}\ S\ \{Q\wedge\neg B\}}$$

Repeat statements

$$\frac{\{P\}S\{Q\},\ Q\wedge\neg B\Rightarrow P}{\{P\}\ \underline{\text{repeat}}\ S\ \underline{\text{until}}\ B\ \ \{Q\wedge B\}}$$

Note that PASCAL allows a sequence of statements to occur between the brackets repeat and until. Thus S stands here for a sequence of statements.

For statements

$$\frac{\{(a\leq x\leq b)\wedge P([a..x])\}S\{P([a..x])\}}{\{P([])\}\ \underline{\text{for}}\ x := a\ \underline{\text{to}}\ b\ \underline{\text{do}}\ S\ \{P([a..b])\}}$$

The notation [u..v] is used to denote the closed interval u..v, i.e. the set $\{i \mid u\leq i\leq v\}$, and [u..v) is used to denote the open interval u...v, i.e. the set $\{i \mid u\leq i<v\}$. Similarly

(u..v] denotes the set $\{i \mid u<i\leq v\}$. Note that [u..u) = (u..u] = [ ] is the empty set.

$$\frac{\{(a\leq x<b)\wedge P((x..v])\}S\{P([x..b])\}}{\{P([])\}\ \underline{\text{for}}\ x := b\ \underline{\text{downto}}\ a\ \underline{\text{do}}\ S\ \{P([a..b])\}}$$

With statements

$$\frac{\{P^{r.s_1\ldots r.s_m}_{s_1\ldots s_m}\}S\{Q^{r.s_1\ldots r.s_m}_{s_1\ldots s_m}\}}{\{P\}\ \underline{\text{with}}\ r\ \underline{\text{do}}\ S\ \{Q\}}$$

$s_1\ldots s_m$ are the field identifiers of the record type of r. Note that r must not contain any variables subject to change by S, and that

$$\underline{\text{with}}\ r_1,\ r_2\ \ldots\ r_n\ \underline{\text{do}}\ S$$

stands for

$$\underline{\text{with}}\ r_1\ \underline{\text{do}}\ \underline{\text{with}}\ r_2\ \underline{\text{do}}\ \ldots\ \underline{\text{with}}\ r_n\ \underline{\text{do}}\ S$$

Monitor axioms (for partial correctness)

If m is a monitor containing the condition variable c and I is the monitor invariant, C is the boolean condition associated with the condition variable c, involving the variables of the monitor m, then

$$\{I\wedge\neg C\}\ \text{c.wait}\ \{I\wedge C\}$$

$$\{I\wedge C\}\ \text{c.signal}\ \{I\}$$

If p is a procedure of the monitor m, that can be

invoked from outside of m, then

$$\frac{\{I \wedge P\}\ p(x,y)\ \{I \wedge Q\}}{\{P\}\ m.p(x,y)\ \{Q\}} \quad .$$

APPENDIX B

VERIFICATION OF THE SIMPLE SCHEDULER SPECIFICATION

In this appendix, we will present the proofs for the programs in the simple scheduler, using the axioms and rules of inference of PASCAL [Hoare and Wirth 1972], which are listed in appendix A. In section B1, the verification of the abstract data type distinct element queue (henceforth called queue, for short) is given. The queues are used extensively in the simple scheduler programs, and in their verification we will use the properties proved in B1. In section B2, the programs of the simple scheduler are verified. In section B3, the theorems stated in section 4.3 are proved.

B1. THE ABSTRACT DATA TYPE DISTINCT ELEMENT QUEUE

The type is defined as: type T = deq[N] of D; where N is a positive integer and denotes the maximum allowable length of the queue and D is the domain of the elements of the queue.

The axioms for the type T are the following:

1. T() is a T. {that is, the empty queue belongs to this type};
2. If x is a T and d is a D and num(x) < N and d is not in x then (x @ d) is a T.
3. These are the only elements of T.
4. num(T() = 0 and for all d in D ¬(d in T()).
5. If x is a T and (num(x) = n < N) and d is not in x then num(x @ d) = n+1 and for all i(1 <= i <= n): x.i = (x @ d).i and (x @ d).n+1 = d.
6. If x is a T and ¬(x = T()) and num (x) = n then num(rest(x)) = n-1 and for all i(1 <= i < n): rest(x).i = x.(i+1).

An element of T is called a queue.

Theorem B1-1:

If x is a T and num(x) = n < N and there exists i,j(1 <= i,j <= n ∧ x.i = x.j) then i = j.

Proof:

The proof is by induction on the length of the queues. The theorem is obviously true for the empty queue T(). Assume it is true for all queues of length <= n, 0 <= n <= N. The only way of generating queues is by the append operation "@".

Let x be any queue of length n. Let d be any element of D not in x. By axioms 2 and 5, x @ d is a T and num(x @ d) = n+1. Thus we have to show the theorem is true for (x @ d). Assume the theorem is false, then the following three cases may arise:

(1) 1 <= i,j <= n.

(2) 1 <= i <= n and j = n+1.

(3) 1 <= j <= n and i = n+1.

In case(1) x.i = x.j => the theorem is false for the queue x, contradicting the induction hypothesis.

In cases(2) and(3) x.n+1 = d = an element of x which contradicts the requirement of axiom 2 that d be not in x.

Thus the theorem is proved.

Theorem B1-2.

If x is a T and $\neg$(x = T()) then rest(x) is a T.

Proof:

By construction, that is, we will show that rest(x) can be constructed from the empty queue T() by the use of axiom 2.

Let num(x) = n and 1 <= n <= N. Let x.i = d_i for all i(1 <= i <= n $\wedge$ d_i in D). By theorem B1-1 for all i,j (1 <= i,j <= n $\wedge$ $\neg$(i = j)) $\neg$(d[i] = d[j]). By

axiom 6, rest(x) = (...(t() @ d-2)... d-n-1).
(...(T() @ d-2) ... d_n-1) satisfies the axiom 2, hence rest(x) is a T.

Theorem B1-3:

If x is a T and ¬(x = T()) then ¬(x.1 in rest(x)).

Proof:

By axiom 6 and theorem B1-1.

Theorem B1-4:

If x is a T and ¬(X = T()) and num(x) = n < N and ¬(d in x) and d in D then rest(x @ d) = rest(x) @ d.

Proof:

Let y = rest(x @ d) and let z = rest(x) @ d then we have to prove that num(y) = num(z) and for all i(1 <= i <= num(y)) y.i = z.i.

Let num(x) = n and 1 <= i < N. num(x @ d) = n+1, by axiom 5. num(rest(x @ d)) = n, by axiom 6. num(rest(x)) = n-1, by axiom 6. num(rest(x) @ d) = n, by axiom 5. Hence num(y) = num(z).

By axiom 5 (x @ d).i = x.i for all i (1 <= i <= n) and (x @ d).n+1 = d. By axiom 6, y.i = x.i+1 for all i (1 <= i < n) and y.n = d.

By axiom 6, rest(x).i = x.i+1 for all i (1 <= i < n).

By axiom 5, z.i = x.(i+1) for all i(1 <= i < n) and z.n = d.

Hence y = z and the theorem is proved.

B1.1. Implementation of type T

The implementation of type T is given in figure B.1.

We prove that the implementation satisfies the axioms for type T as follows:

Let s be of type T, then using the class notation [Dahl and Hoare 1972] we wish to prove the following:

$$s = T() \text{ iff } s.is_empty = true \quad (1)$$

$$(x=T()) \Rightarrow s.1 = s.first \quad (2)$$

$$\{\neg(s=T()) \wedge s = s0\}\ s.remove\ \{s = rest(s0)\} \quad (3)$$

$$\{num(s) < N \wedge (d \in s) \wedge s = s0\}s.append(d)\{s=s0 \oplus d\} \quad (4)$$

Proof:

The queue is represented by the following relationship between the abstract variables and the concrete variables:

$(t = 0 \Rightarrow (s = T() \wedge num(s) = 0))$ and

$(\neg(t = 0) \Rightarrow t = s.1 \wedge a[s.i] = s.i+1,\ 1 <= i < num(s))$

and $(num(s) > 0 \Rightarrow a[s.num(s)] = 0)$.

This is known as the representation relation.

Using the above relationship we have to show that initially the queue is empty and that further the four

```
class T: deq[N] of 1..N;
begin
var t : 0..N;
    a : array[1..N] of 0..N;

function first: 1..N;
begin if (t ≠ 0) then first := t; end first;

function is_empty : Boolean;
begin is_empty := t = 0; end is_empty;

procedure remove;
begin if (t ≠ 0) then t := a[t]; end remove;

procedure append (d : 1..N);
var x : 1..N;
begin a[d] := 0;
   if t = 0 then t := d
   else begin
           x := t;
           while (a[x]≠0) do x := a[x];
           a[x] := d;
        end;
end append;

{initially} t := 0;  {the queue is initialized to be empty};
end class T;
```

Figure B.1

Implementation of Type T (distinct element queue)

relations stated above hold.

After initialization t := 0{t=0} is true and by the relationship between the concrete variables and the abstract object, s = T().

To show that is_empty = (s=T()), note that (s=T()) = (t=0) and {true} is_empty := t = 0 {is_empty = (t=0)}. Hence is_empty = (s=T()).

To show that s.1 = first, note that s.1 is defined only when (s ≠ T()), that is only when (t ≠ 0), and {(t ≠ 0} first := t{first = t}∧t = s.1 by the representation relation.

To show that {s=s0 ∧ s ≠ T()} remove {s = rest(s0)} we have to show, by axiom 6, that {s=s0 ∧ s≠T()} remove {s.i=s0.i+1, 1<=i<num(s) ∧ num(s) = num(s0)-1}. Note that (s≠T()) implies (t≠0) and {(t≠0) ∧ (t=t0)} t := a[t]{t = a[t0]}, thus using the representation relation and considering the case when num(s0) = 1, we have num(s) = num(s0)-1 = 0 and t = a[t0] = a[s0.num(s0)] = 0. In the case num(s0) > 1, we have s.1 = t = a[t0] = s0.2 and ∀i(1 < i < n) s.i = s[s.i-1] = a[s0.i] = s0.i, by induction and s[s.n-1] = a[s0.n] = 0, since the array, a, is unchanged.

To show that {(d ∉ s)∧dε1..N}append(d){s=s0 ⊕ d},

note that (d $\notin$ s) and d $\in$ 1..N implies num(s0) = n < N (by theorem B1, since all elements of s are distinct and in 1..N). There are two cases, when s = T() and when s $\neq$ T(). Consider the case when s = T(), then append reduces to

```
begin a[d] := 0; t := d; end;
```

That is, {t = 0 and t = t0 and a = a0}

```
begin a[d] := 0; t := d; end;
{(a0,d:0) ∧ t=d} .
```

Thus s.1=d and num(s) = 1 for a[d] = 0; which is what is required to be proved by axiom 5.

In the case when (s $\neq$ T()), we have to show that

{t = t0 $\wedge$ a = a0 $\wedge$ (d $\notin$ s)}

```
begin a[d] := 0;
      x := t;
   while   (a[x] ≠ 0) do x := a[x];
   a[x] := d;
end;
```

$\forall$i(1 <= i <= n) s.i = s0.i $\wedge$ s.n+1 = d $\wedge$ num(s) = n+1.

The proof follows from the following assertions:

{a = a0 $\wedge$ (d $\notin$ s0)} a[d] := 0{a = (a0,d:0) $\wedge$ $\forall$i(1 <= i <= n) s.i = s0.i

{t = s.1} x := t{x = s.1 $\wedge$ t = s.1}

{x = x.1 $\wedge$ 1 <= i <= n $\wedge$ (a[x] $\neq$ 0)} x := a[x]

{x = s.i $\wedge$ 1 <= i <= n $\wedge$ i = (i0+1)}

Thus {x = s.1} while (a[x] ≠ 0) do x := a[x];
{x = s.n ∧ s = s0 ∧ a = a0} and {x=s.n ∧ t=t0 ∧ s=s0 ∧
num(s) = n} a[x] := d; {s.n+1 = d ∧ num(s) = n+1 ∧
a = ((a0,d:0),s.n:d) ∧ t = t0}.

The proof of termination follows from the fact that there are no loops except for the while statement in the procedure append, and its proof of termination is given below.

{x = s.i0 ∧ i <= i0 <= num(s) ∧ (a[x] ≠ 0)}
x := a[x]
{x = s.i ∧ i = i0+1 ∧ i <= i <= num(s)}

Since i = i0+1 implies num(s)-1 < num(s)-i0, which is a well-founded ordered function. The proof of termination follows from the well-founded ordered function termination rule [Manna 1974]. Thus the termination is proved with respect to the representation relation.

The advantage of using the class definition is that we are assured that the variables cannot be altered by any other procedures. The proof that the representation and the procedures implement a queue will still be valid if we had some other means of assuring that the variables are not altered. Thus if we had passed the variables as parameters and asserted that they represent a queue then the output assertion of the procedures and functions would

still be valid. The disadvantage of using class definitions is that we cannot share data among many instances of the class. Thus if we had a set of queues which satisfy the constraint that the intersection of any two queues in the set is empty then we can implement this set of queues with a shared array instead of an array per queue. Note that the number of queues in this set may vary dynamically as long as they satisfy the constraint. The assertion that they can share the same array can be verified as follows:

Let scope (x) where x is a procedure or function name be the set of variables accessed by the procedure. Then

scope(first) = t

scope(is_empty) = t

scope(remove) = (t,a[t])

scope(append(d)) = (t, the set {a[i] | i in s}, a[d],d)

Thus it is obvious that if the queues are disjoint then when the functions or procedures are invoked then their scopes are always disjoint as far as the array elements are concerned, hence we can use the same array without any conflict.

B2. VERIFICATION OF THE SIMPLE SCHEDULER PROGRAMS.

In this section we will verify the programs for the procedures exit, enter, wait, and signal. As noted in chapter 3 a process can be in one of three possible states: active, ready or blocked. The distinction between active and ready in relevant only to the simple scheduler and not to any of the higher levels. This is so, because the sharing of the processors among processes, is to be invisible to higher levels. Hence in our assertions, we will not distinguish between these two states and consider a process to be in either blocked or unblocked state. The procedure dispatch, which is responsible for allocating a processor to a ready process will be considered to be of no effect. It is assumed that when a blocked process is unblocked it resumes execution at the statement following the one that caused it to block, considering the calls on the simple scheduler procedures as single primitive statements.

We will use the properties of queues proved in section B1. As noted at the end of section B1, we will use a global array (PLINK) for implementing the queues for all monitors and condition variables. The queues for all monitors and condition variables satisfy the disjointness

property, since no process can be in more than one queue at any time.

Note that in the specification of the procedures enter, exit, wait and signal, the code is bracketed by "enter_mutual_exclusion_state" and "exit_mutual_exclusion_state". It is assumed that by thus bracketing (and providing a basic mutual exclusion mechanism in the hardware) we are assured of sequential execution of the simple_scheduler procedures and we can assume the concatenation axiom for all the statements in the programs. These statements are omitted in the programs given below.

The verification of the functions first and is_empty as well as the procedures append and remove is not given in this section because they have been verified in section B1. The verification of the procedures signal_interrupt and wait_interrupt is not given because it is very similar to that of the procedures exit and enter respectively.

The proof of termination is obvious because there are no loop statements and all procedures and functions called are either proved to be terminating or assumed to terminate.

The verification of the procedures enter, exit, wait, and signal is given in figure B.2.

```
{declarations for the simple scheduler}
type processid = 1..N; {N, an integer, is the number of
                        processes at this level;}
     monitorname = record in_use : Boolean;queue : 0..N
                   end;
     condition = 0..N; {0 is used as the null value for
                        queue and condition};
var plink : array [processid] of 0..N; {processid, null};
    status : array [processid] of (active, ready, blocked);

Proof of the procedure enter

procedure enter (var m: monitorname; p: processid);
INPUT ASSERTION: status[p] ≠ blocked ∧ m=m0 /m0 is a free
                                              variable/
OUTPUT ASSERTION: (¬m0.in_use => m.in_use ∧ status[p] ≠
                                            blocked) ∧
                  (m0.in_use => m.in_use ∧ status[p] =
                   blocked ∧ m.queue = m0.queue ⊖ p)
begin
   if ¬(m.in_use) then
   {status[p] ≠ blocked ∧ m=m0 ∧¬m.in_use}
        m.in_use := true
   {¬m0.in_use => m.in_use ∧ status[p] ≠ blocked}
   else
   {status[p] ≠ blocked ∧ m=m0 ∧ m.in_use}
      begin
         append(m.queue,p)
         {m0.in_use => m.in_use ∧ status[p] ≠ blocked ∧
           m.queue = m0.queue ⊖ p}
         status[p] := blocked;
         {m0.in_use => m.in_use ∧ status[p] = blocked ∧
           m.queue = m0.queue ⊖ p}
         dispatch(p);
      end;
end enter;
{(¬m0.in_use => m.in_use ∧ status[p] ≠ blocked) ∧
 (m0.in_use => m.in_use  ∧ status[p] = blocked ∧
               m.queue = m0.queue @ p)}

End of proof of the procedure enter
```

Figure B.2

Verification of the Simple Scheduler

```
Proof of the procedure exit
procedure exit (var m: monitorname; p: processid);
INPUT ASSERTION: status[p] ≠ blocked ∧ m.in_use ∧
                 m.queue = m0.queue
OUTPUT ASSERTION: (m0.queue=empty => ¬m.in_use ∧
                   m.queue = empty ∧ status[p] ≠ blocked) ∧
                  (m0.queue ≠ empty => status[p]≠blocked ∧
                   m.queue = rest(m0.queue) ∧ m.in_use ∧
                   status[first(m0.queue)] ≠ blocked)
begin
   if is_empty(m.queue) then
   {status[p] ≠ blocked ∧ m.in_use ∧ m.queue = m0.queue ∧
                          m.queue = empty}
      m.in_use := false
   {m0.queue=empty => ¬m.in_use ∧ m.queue = empty ∧
                      status[p] ≠ blocked}
   else
   {status[p] ≠ blocked ∧ m.in_use ∧ m.queue = m0.queue ∧
                          m.queue ≠ empty}
      begin
         status[first(m.queue)] := ready;
         {m0.queue ≠ empty ∧ m.in_use ∧ status[p] ≠
          blocked ∧ m.queue ≠ empty ∧
          status[first(m0.queue)] ≠ blocked}
          remove (m.queue);
         {m0.queue ≠ empty ∧ m.in_use ∧ status[p] ≠
          blocked ∧ m.queue = rest(m0.queue) ∧
          status[first(m0.queue)] ≠ blocked}
          status[p] := ready;
         {m0.queue ≠ empty => (m.in_use ∧ status[p] ≠
          blocked ∧ m.queue = rest(m0.queue) ∧
          status[first(m0.queue)] ≠ blocked)}
          dispatch(p);
      end;
end exit;
{(m0.queue = empty => ¬m.in_use ∧ m.queue = empty ∧
                      status[p] ≠ blocked) ∧
 (m0.queue ≠ empty => (m.in_use ∧ m.queue=rest(m0.queue) ∧
  status[p] ≠ blocked ∧ status[first(m0.queue)] ≠ blocked))}
End of proof of the procedure exit
```

Figure B.2 (continued)

```
Proof of the procedure wait
procedure wait (var m: monitorname; var cv: condition;
                p: processid);
INPUT ASSERTION: status[p] ≠ blocked ∧ m.in_use ∧
                m.queue = m0.queue ∧ cv = cv0
OUTPUT ASSERTION: status[p] = blocked ∧ cv = cv0 @ p ∧
                (m0.queue = empty =>¬m.in_use ∧
                          m.queue = empty) ∧
                (m0.queue ≠ empty => m.in_use ∧
                 m.queue = rest(m0.queue) ∧
                 status[first(m0.queue)] ≠ blocked)
begin
   append (cv,p);
   {status[p] ≠ blocked ∧ m.in_use ∧ m.queue = m0.queue ∧
                          cv = cv0 @ p}
   status[p] := blocked;
   {status[p] = blocked ∧ cv=cv0 @ p ∧ m.in_use ∧
    m.queue = m0.queue}
   if is_empty(m.queue) then
   {status[p] = blocked ∧ cv = cv0 @ p ∧ m.in_use ∧
    m.queue = m0.queue ∧ m.queue = empty}
      m.in_use := false
   {status[p] = blocked ∧ cv = cv0 @ p ∧
    (m0.queue = empty =>¬m.in_use ∧ m.queue = empty)}
```

Figure B.2 (continued)

AD-A040 699 STANFORD UNIV CALIF STANFORD ELECTRONICS LABS F/G 9/2
A VERIFIED SPECIFICATION OF A HIERARCHICAL OPERATING SYSTEM.(U)
JAN 76 A R SAXENA N00014-75-C-0601
UNCLASSIFIED TR-107 NL

3 OF 3
AD A040699



END
DATE FILMED
7-77

```
    else
    {status[p] = blocked ∧ cv = cv0 @ p ∧ m.in_use ∧
     m.queue = m0.queue ∧ m.queue ≠ empty}
       begin
          status[first(m.queue)] := ready;
       {status[p] = blocked ∧ cv = cv0 @ p ∧
        m0.queue = m.queue ∧ m0.queue ≠ empty ∧ m.in_use ∧
        status[first(m0.queue)] ≠ blocked}
           remove (m.queue);
       {status[p] = blocked ∧ cv = cv0 @ p ∧ (m0.queue ≠
        empty => m.in_use ∧ m.queue = rest(m0.queue) ∧
        status[first(m0.queue)] ≠ blocked)}
        end;
    dispatch(p);
end wait;
{status[p] = blocked ∧ cv = cv0 @ p ∧
 (m0.queue = empty => m.in_use ∧ m.queue = empty) ∧
 (m0.queue ≠ empty => m.in_use ∧ m.queue = rest(m0.queue) ∧
  status[first(m0.queue)] ≠ blocked}
End of proof of the procedure wait
```

Figure B.2 (continued)

Verification of the Simple Scheduler

```
Proof of the procedure signal
procedure signal (var m: monitorname; var cv: condition;
                  p: processid);
INPUT ASSERTION: status[p] ≠ blocked ∧ m.in_use ∧
                 m.queue = m0.queue ∧ cv = cv0
OUTPUT ASSERTION: (cv0 = empty =>(status[p] ≠ blocked ∧
                  m.in_use ∧ m.queue = m0.queue ∧ cv=cv0))∧
                 (cv0 ≠ empty =>(status[p] = blocked ∧
                  m.in_use ∧ m.queue = m0.queue @ p ∧ cv =
                  rest(cv0) ∧ status[first(cv0)]≠blocked))
begin
   if ¬(is_empty(cv)) then
   {status[p] ≠ blocked ∧ m.in_use ∧ m.queue = m0.queue ∧
    cv = cv0 ∧ cv ≠ empty}
     begin
        append(m.queue,p);
        {cv0 ≠ empty ∧ status[p] ≠ blocked ∧ m.in_use ∧
         m.queue = m0.queue @ p ∧ cv = cv0}
         status[p] := blocked;
        {cv0 ≠ empty ∧ status[p] = blocked ∧ m.in_use ∧
         m.queue = m0.queue @ p ∧ cv = cv0}
        status[first(cv)] := ready;
        {cv0 ≠ empty ∧ status[p] = blocked ∧ m.in_use ∧
         m.queue = m0.queue @ p ∧ cv = cv0 ∧
         status[first(cv)] ≠ blocked}
        remove (cv);
        {cv0 ≠ empty => status[p] = blocked ∧ m.in_use ∧
         m.queue = m0.queue @ p ∧ cv = rest(cv0) ∧
         status[first(cv)] ≠ blocked}
        dispatch;
     end;
end signal;
{(cv0 = empty => status[p] ≠ blocked ∧ m.in_use ∧
                 m.queue = m0.queue ∧ cv = cv0 ∧
 (cv0 ≠ empty => status[p] = blocked ∧ m.in_use ∧ m.queue =
                 m0.queue @ p ∧ cv = rest(cv0) ∧
                 status[first(cv)] ≠ blocked}
End of proof of the procedure signal
```

Figure B.2 (continued)

Verification of the Simple Scheduler

B3. PROOF OF CORRECTNESS OF THE SIMPLE SCHEDULER THEOREMS

In this section we will prove the properties of the simple scheduler mentioned in chapter 4. The properties deal with access to a monitor and are proved subject to the following assumptions. The theorems P1, P2, P4 and the corrollary P3 correspond to the properties P1, P2, P4 and P3 given in section 4.3).

A1. A process p is said to 'have access to a monitor' m if and only if it has completed a call on the procedure enter, with actual parameters m and p or completed a call on the procedures wait or on the procedure signal with actual parameters m, cv, p, where cv is a condition variable local to the monitor m.

A2. A process, p, which currently has access to the monitor m, no longer has access if it calls the procedure exit with actual parameters m and p, or if it has called the procedures wait or procedure signal and has not yet completed the call.

A3. A process p, calls enter (m,p), only if it does not currently have access to the monitor m.

A4. A process p calls on procedures exit (m,p), wait (m,cv,p), and signal (m,cv,p) only if it has current

access to the monitor m.

A5. A call by a process p, is completed only if the procedure terminates and the process p is not in a blocked state, that is status[p] $\neq$ blocked.

A6. A blocked process cannot execute any program statements.

Theorem P1:

At any time, at most one process has access to a monitor m.

Proof:

The theorem is proved by demonstrating the validity of the following invariant.

m.in_use $\equiv$ exactly one process has access to the monitor m $\wedge$

$\neg$(m.in_use) $\equiv$ no process has access to the monitor

Initially m.in_use is false and no process has access to the monitor, thus the invariant is true.

Assuming the invariant is true prior to a call on the procedure enter (m,p), the call on enter can be divided into two cases: if initially m.in_use is true, then by the invariant there is already a process other than p, with current access to the monitor m, and as the output assertion (proved in B2) states, m.in_use is true at the

output and p is in a blocked state. Thus, process p is not given access to the monitor m, satisfying the invariant; if initially m.in_use is false then no process has access to the monitor m, and as the output assertion states, m.in_use is true and process p is given access to the monitor m, satisfying the invariant.

Assuming the invariant is satisfied prior to exit (m,p) and wait (m,cv,p), by assumption A3, m.in_use has to be true, and on termination of the procedures either m.in_use is false and no process is waiting to enter the monitor m (m.queue = empty, note that a process joins the queue only after an uncompleted call on enter or signal), and hence no process currently has access to the monitor m, satisfying the invariant or m.in_use is true and exactly one of the waiting processes is given access to the monitor. (proved input/output assertions of procedures exit, wait in B2). Note that by assumption A2, process p, which had access to m, no longer has access to m.

Assuming the invariant is satisfied prior to call on signal (m,cv,p), by A4, m.in_use is true and only process p has access to m. If no process is waiting on the condition variable cv (cv = empty), then the effect of signal is null (proved in B2) and thus the invariant is satisfied.

If at least one process is waiting on the condition variable cv, (note that a process x joins the queue cv, only on executing wait (m,cv,x)) then the process p is blocked and hence releases access to m, also exactly one of the processes waiting on cv is unblocked, (completing its call on wait) and given access to m. Since m.in_use is not changed the invariant is satisfied.

Since the invariant is satisfied initially and also at the termination of the procedures enter, exit, wait and signal (assuming the invariant at their initiation) the theorem is proved.

Theorem P2:

No process p is denied access to a monitor m forever on calling enter (m,p) assuming that a process that currently has access to the monitor m, eventually releases the access, i.e. calls exit or wait.

Proof:

There are three cases to be considered.

(1) There is currently no process that has access to m and a process p is denied access.

(2) There is currently a process x, that has access to m and p is the only process waiting to enter the monitor m.

(3) There is currently a process x, that has access to m and p is one of many processes waiting to enter the monitor m.

The case (1) is impossible as is obvious from the output assertion of enter (proved in B2).

In case (2), p = first (m.queue) and when process x executes a wait or an exit to release access, by the output assertion of exit and wait, p will gain access to the monitor m. If x executed a signal then either x retains access or some other process gains access to m. Assuming a finite number of processes, no processes other than those already waiting on a condition variable can gain access to m before p. Since there are only a finite number of processes waiting on condition variables, by the theorem's hypothesis one of them has to execute the procedure wait or exit that will enable p to gain access to the monitor m.

In case (3), every time a process waiting to enter m, gains access to m, p advances in the waiting line (by the properties of queues proved in B1) and eventually p will be first (m.queue), in which case, the situation reduces to that of case 2.

Hence the theorem is proved. (We can also show that

m.in_use = false ∧ m.queue ≠ empty is not possible, by an inspection of the input/output assertions of the procedure enter, exit, wait and signal and after initialization. Thus assuming every process that has access to the monitor eventually calls exit and using the FIFO properties of the queue, it is clear that no process is denied access to m forever).

Theorem P3a:

A process p, calling wait (m,cv,p) is blocked and added to the queue for cv.

Proof:

Follows from the verification of procedure wait in B2.

Theorem P3b:

A process p currently first on the queue cv of monitor m and blocked, is unblocked only by a signal operation on cv by another process and gains immediate access to the monitor m.

Proof:

Follows from the verification of the procedure signal in B2.

Corollary P3:

When a process executes a wait operation on a condition variable, it is resumed only after a subsequent signal operation on the same condition variable, and when a process is so resumed, no other process gains access to the monitor after the signal operation (i.e. whatever is the state of the monitor before the initiation of the signal will be the state found by the resumed process, thus at the termination of the wait).

Proof:

Follows from theorems P3a and P3b.

Theorem P4:

A process calling signal (m,cv,p) is blocked only if a process is waiting on the queue associated with the condition variable cv, and in this case will be resumed when no other process is in the monitor (i.e. when some other process executes a wait or an exit operation).

Proof:

Follows from the verification of the procedure signal since the process is blocked and put on the queue associated with the monitor m only if the queue associated with cv is nonempty, otherwise the operation has no effect. The process is unblocked only by the procedures wait or exit

and in this case no other process will be in the monitor (proved in B2).

Proof of process assertions for monitor primitives

As stated in chapter 4, there is a difference between the verification of a program and the verification of a program's execution on behalf of a process. The verification of the monitor primitives from the viewpoint of the process invoking them is given below.

Let process_in (m) be a variable (auxiliary) denoting the process having exclusive access to the monitor m. Let I(m) be the invariant associated with the variables of the monitor m that is satisfied after initialization and on exit from the monitor as well as being a precondition for all wait operations. Let C be the Boolean condition associated with the condition variable c. Let all preconditions of signal operations on c imply the Boolean condition C and the invariant I. We can, given the above assumptions, prove the following assertions from the viewpoint of a process invoking the monitor primitives:

(1) {process_in (m) $\neq$ p}

enter (m,p)

{process_in (m) = p $\wedge$ I(m)}

(2) {process_in (m) = p ∧ I(m)}

exit (m,p)

{process_in (m) ≠ p}

(3) {process_in (m) = p ∧ I(m)}

wait (m,c,p)

{process_in (m) = p ∧ I(m) ∧ C}

(4) {process_in (m) = p ∧ I(m) ∧ C}

signal (m,c,p)

{process_in (m) = p ∧ I(m)}

The verification of assertion (1) follows from the verification of the procedure enter, the theorems P1 and P2, and the assumption A1.

The verification of assertion (2) follows from the verification of the procedure exit and the assumption A2.

The verification of assertion (3) follows from the corollary P3 and the precondition for the procedure signal.

The verification of assertion (4) follows from the theorem P4 and the verification of the procedures exit and wait.

APPENDIX C

VERIFICATION OF THE SIMPLE MEMORY MANAGER SPECIFICATION

In this appendix we will present the verification of the simple memory manager specification. The verification of the specification is presented in two steps. First the partial verification is given in section C1, i.e. without the proof of termination and absence of deadlocks. In section C2, the proof of termination and the absence of deadlocks is given. In the verification we have made use of the notion of exclusive access and concepts for demonstrating absence of deadlocks developed in section 4.2. We have also made use of the monitor axioms stated in section 4.3. In section C1 we have also proved some properties of the process automatic_discard.

The resources to which a monitor has exclusive access are expressed in the invariant relation. When the input assertion of a monitor procedure states exclusive access to any resource, then the interpretation is that the process invoking the monitor procedure is granting exclusive access to the monitor for those resources (of course the process must have exclusive access to those resources

at the time of invoking the procedure) and if the output assertion of a monitor procedure states any exclusive resource, then the interpretation is that the monitor is granting exclusive access to the process invoking the procedure. The monitor maintains exclusive access during the execution of the procedure unless it invokes some other monitor's procedure and releases access (it may also gain access to resources by these means).

Another difference between concurrent and sequential programs is that the state of the system or data structures may be nondeterministic in the case of concurrent programs. For example, when the procedure fetch is invoked we do not know what the state of the vm will be. There could be more than one fetch operation invoked simultaneously, although only one will take precedence. Thus the meaning of the assertion vm = vm0 is unclear. Of course, in circumstances like these, we are interested only in the state of the vm just before the operation is actually invoked (in this case by the monitor address_-mapper). So for re-entrant procedures, and in general assertions regarding data controlled by monitors, we will assume the input assertion refers to values on entering the critical section and the output assertion refers to values

at the time of leaving the critical section. Observe that these may be quite different from the values, before the critical section is entered but after an attempt is made and after exiting from the monitor.

In our assertions we have used the following abbreviations (predicates or functions).

H(v) is a nonnegative integer function. It is the number of times the procedure set_map of the monitor address_mapper is invoked with v as a parameter. (It is the number of times the page is brought into the main memory).

G(v) is a predicate. If it is in the output assertion of a procedure then it implies that there existed an integer m, $0 \leq m < M$ such that m_adr[m] = v in the data structure of the monitor address_mapper, during the execution of the procedure, i.e. the virtual page v was in the main memory.

F(m) is a predicate. If it is in the output assertion of a procedure then it implies that the main memory page frame m was added to (or is in the pool of free main memory page frames, i.e. added to m_pool (Note that on termination of the procedure it may no longer be in m_pool).

The predicate IAD(v) is true if and only if there exists an m such that m_adr[m] = v in the data structure of the monitor address_mapper.

C1 PARTIAL VERIFICATION OF THE SIMPLE MEMORY MANAGER SPECIFICATIONS

In this section we present a partial verification (that is, we do not verify termination and absence of deadlocks) of the simple memory manager specification. We have not presented the verification of the procedure assign because it is very similar to that of the procedure fetch, and of mm_assign (in the monitor address_mapper) because its verification is very similar to that of the procedure mm_fetch. We have also not presented the verification of the monitor sm_alloc because it is very similar to that of the monitor mm_alloc. The verification is given in figure C.1. The notation $ex(r_1,\ldots,r_n)$ is used to denote exclusive access to the resources $r_1,\ldots,r_n$.

```
Proof of the procedure fetch

procedure fetch (va: virtual_address; var w: word);

INPUT ASSERTION: true

OUTPUT ASSERTION: w = vm[va] ∧ vm = vm0
var pf: Boolean;
begin
    address_mapper.mm_fetch(va,w,pf);
   {vm = vm0 ∧ H(va.vpn0) = h0 ∧ ( ¬pf => w=vm[va])}
   /follows from the proof of address_mapper.mm_fetch/
    while pf do
    {H(va.vpn0) = h1 ∧ h1 ≥ h0 ∧ pf}
    /follows from the precondition for the use of while
      axiom/
       begin
          page_fault(va.vpn0);
          {H(va.vpn0) > h1 ∧ G(va.vpn0) ∧ vm = vm0}
          /follows from the proof of page_fault/
          address_mapper.mm_fetch(va,w,pf)
          {vm = vm0 ∧ H(va.vpn0) = h0 ∧ ( ¬pf => w=vm[va])}
          /follows from the proof of
           address_mapper.mm_fetch/
       end
          {vm = vm0 ∧ (¬¬pf => w=vm[va]) ∧ ¬¬pf)}
          /follows from the use of while axiom/
end fetch;
    {vm = vm0 ∧ w=vm[v]}
    /follows from the logical simplification of the assertion
       above/.

End of proof of the procedure fetch
```

Figure C.1

Partial Verification of the Simple Memory Manager

```
Proof of procedure page_fault

procedure page_fault(v: vpf);

INPUT ASSERTION: H(v) = h0

OUTPUT ASSERTION: H(v) > h0 ∧ G(v)

var m: mmpf; s: smpf; st: (in,out,all_zero);
begin
   page_fault_handler.lock_for_input(v,s,st);
   {(st = in => ((H(v) > h0) ∧ G(v)) ∧
    (st = out => (vm[v] = sm[s]) ∧ ex(vm[v],sm[s])) ∧
    (st = all_zero => (vm[v]=0) ∧ ex(vm[v])) }--A1
    /follows from the proof of
     page_fault_handler.lock_for_input/
      case st of
         in: {st = in ∧ A1}
             {do nothing}; {H(v) > h0 ∧ G(v)}
             /follows from the input assertion to the case
              statement and the case st = in/.
      out: {st = out ∧ A1}
         begin
             mm_alloc.mm_acquire(m);
             {ex(mm[v]) ∧ st = out ∧ A1 ∧ sm[s] = s0}
             /follows from the proof of
              mm_alloc.mm_acquire/
             input(m,s);
             {A1 ∧ st = out ∧ ex(mm[v]) ∧ mm[m] = sm[s]
                          ∧ sm[s] = s0}
             /follows from the output assertion of input/
             page_fault_handler.bringin(v,m,s);
             {H(v) > h0 ∧ G(v)}
             /follows from the proof of
              page_fault_handler.bringin/
         end;
      all_zero: {A1 ∧ st = all_zero}
         begin
             mm_alloc.mm_acquire(m);
             {A1 ∧ st = all_zero ∧ ex(mm[m])}
```

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

```
                    {A1 ∧ st = all_zero ∧ ex(mm[m])}
                    sm_alloc.sm_acquire(s);
                    {A1 ∧ st = all_zero ∧ ex(mm[m],sm[s])}
                    /follows from proof of sm_alloc.sm_acquire/
                    mm[m] := 0;
                    {A1 ∧ st = all_zero ∧ ex(mm[m],sm[s]) ∧
                        mm[m]=0 ∧ mm[m] = vm[v]}
                    /follows from assignment axiom and the lemma
                       mm[m] = 0 ∧ vm[v] = 0 => mm[m] = vm[v]/
                    page_fault_handler,bringin(v,m,s);
                    {H(v) > h0 ∧ G(v)}
                end;
            end;
               {H(v) > h0 ∧ G(v)}
               /follows from case axiom/
end page_fault;
   {H(v) > h0 ∧ G(v)}

End of proof of the procedure page_fault
```

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

Proof of the monitor address_mapper

```
monitor address_mapper;
begin
var m_adr: array[mmpf] of 0..v;
INVARIANT: ∀v(0 ≤ v < V) ∧ ∀m(0 ≤ m < M):
           m_adr[v] = v => (vm[v] = mm[v] ∧
             ex(vm[v],mm[v]))
```

Proof of the initialization

INPUT ASSERTION: true

OUTPUT ASSERTION: I.

```
   for m := 0 to M-1 do
       m_adr[m] := V;
   {∀v(0 ≤ v < V) ∧ ∀m(0 ≤ m < M):
    m_adr[m] = v => (vm[v]=mm[m] ∧ ex(vm[v],mm[m]))}
   /follows vacuously from the fact that
    ∀m(0 ≤ m < M) m_adr[m] = V/
```

End of proof of the initialization

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

```
Proof of the procedure mm_fetch

procedure mm_fetch(va: virtual_address; var w: word;
                   var pf: Boolean);

INPUT ASSERTION:  I ∧ vm = vm0

OUTPUT ASSERTION:  I ∧ vm = vm0 ∧ (¬pf => w = vm[va])

var m: 0..M;
begin
   adr_trans(va.vpno,m,pf);
   {I ∧ vm = vm0 ∧ (¬pf => mm[m] = vm[va.vpn0])}
   /follows from the proof of adr_trans/
   if ¬¬(pf) then
      {I ∧ vm = vm0 ∧ (¬¬pf => mm[m] = vm[va.vpno]) ∧¬pf}
      /follows from the if clause/
         w := mm[m, va.offset];
      {I ∧ vm = vm0 ∧ (¬¬pf => w = vm[va])}
      /follows from the assignment axiom, logical simpli-
         fication and the if axiom/
end mm_fetch;
   {I ∧ vm = vm0 ∧ (¬¬pf => w = vm[va]}

End of proof of the procedure mm_fetch
```

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

Proof of the procedure adr_trans

```
procedure adr_trans(v: vpf; var m: 0..M; var pf: Boolean);
INPUT ASSERTION: I ∧ vm = vm0
OUTPUT ASSERTION: I ∧ vm = vm0 ∧ (¬pf => vm[v] = mm[m])
begin
   m := 0;
   {I ∧ vm = vm0 ∧ m = 0}
   /follows from assignment axiom/
   while m < M ∧ ¬(m_adr[m] = v) do
   {I ∧ vm = vm0 ∧ (m < M ∧ ¬(m_adr[m] = v))}
   /follows from while clause/
      m := m+1;
   {I ∧ vm = vm0∧  (m = M ∨ m_adr[m] = v)}
   /follows from while axiom/
      pf := (m = M);
   {I ∧ vm = vm0 ∧ (¬pf ≡ m_adr[m] = v)}
   /follows from the assignment axiom/
end adr_trans;
   {I ∧ vm = vm0 ∧ (¬ pf => mm[m] = vm[v])}
   /follows from logical simplification/
```

End of proof of the procedure adr_trans

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

Proof of the procedure set_map

```
procedure set_map (v: 0..V; m: 0..M-1);

INPUT ASSERTION: i ∧ m_adr = m_adr0 ∧
                 (v ≠ V => (mm[m]=vm[v] ∧
                  ex(mm[m],vm[m]))

OUTPUT ASSERTION: I ∧ m_adr = (m_adr,m:v) ∧
                  (v=V => ex(mm[m],vm[m_adr0[m]]))

begin
   m_adr[m] := v;
   {m_adr=(m_adr,m:v) ∧ I ∧
    (v=V => ex(mm[m],vm[m_adr0[m]]))
   /follows from the array assignment axiom and the
    input assertion/
end set_map;
   {I ∧ m_adr = (m_adr,m: v) =>
    (IAD(v) for 0 ≤ v < V) ∧
    (ex(mm[v],vm[m_adr0[m]]) for v = V)}
```

End of proof of the procedure set_map

End of proof of the monitor address mapper

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

Proof of the monitor mm_alloc

```
monitor mm_alloc;
begin
   var m_pool: set of mmpf;
       m_nonempty: condition;
INVARIANT: ∀m(0 ≤ m < M)  m ∈ m_pool => ex(mm[m])
```

Proof of the initialization

```
INPUT ASSERTION: ∀m(0 ≤ m < M): ex(mm[m])
OUTPUT ASSERTION: I
   m_pool := [0..M-1]
   {m_pool = [0..M-1]}
   /by assignment axiom/
   {m_pool = [0..M-1] ∧ ∀m(0 ≤ m < M)ex(mm[m])
      /by hypothesis/ implies the invariant}
```

End of proof of initialization

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

```
Proof of the procedure mm_acquire

procedure mm_acquire (var m: mmpf);

INPUT ASSERTION: I

OUTPUT ASSERTION: I ∧ ex(mm[m])

begin
   if m_pool = empty then m_nonempty.wait
   {I ∧ m_pool ≠ empty}
   /follows from the wait axiom and the if_clause/
   m := anyoneof(m_pool);
   {I ∧ m_pool ≠ empty ∧ m ∈ m_pool}
   /follows from the assignment axiom and the function
      anyoneof/
   m_pool := m_pool-[m];
   {I ∧ ex(mm[m])}
   /follows from the set subtraction and assignment axiom
    and the use of I to deduce the passing of exclusive
    access/
end mm_fetch;
   {I ∧ ex(mm[m])}

End of proof of the procedure mm_acquire
```

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

Proof of the procedure mm_release

```
procedure mm_release (m: mmpf)

INPUT ASSERTION:  I ∧ ex(mm[m])

OUTPUT ASSERTION:  I ∧ ¬ex(mm[m]) ∧ F(m)

begin
   m_pool := m_pool+[m];
   {I ∧ m ∈ m_pool ∧ (m_pool ≠ empty) ∧ ¬ex(mm[m])
      ∧ F(m)}
   /follows from set addition, assignment axiom.  The
    invariant I follows from the input assertion, i.e. I
    is satisfied for m_pool-[m] and exclusive access is
    granted to m, when m is passed, so I is also satis-
    fied for all of m/
   m_nonempty.signal;
   {I ∧ ¬ex(mm[m]) ∧ F(m)}
   /follows from the signal axiom/
end mm_release
   {I ∧ ¬ex(mm[m]) ∧ F(m)}
```

End of proof of the procedure mm_release

End of proof of the monitor mm_alloc.

Figure C.1 (continued)

Partial Verification of the Simple Memory

Proof of the monitor page_fault_handler

```
monitor page_fault_handler;

begin
type vp_state = record loc(in,out,all_zero); mm_adr:
                  0..M-1; sm_adr: 0..S-1 end;
var virtual_map: array [0..M-1] of 0..V;
   m_adr: array [0..M-1] of 0..V;
   page_inuse: array [0..V-1] of Boolean;
   page_nonbusy: array [0..V-1] of condition;

INVARIANT: ∀v(0 ≤ v < V):
   with virtual_map[v]
      (loc = in => mm[mm_adr] = vm[v]) ∧
      (loc = out => sm[sm_adr] = vm[v]) ∧
      (loc = all_zero => vm[v] = 0) ∧
∀m(0≤m<M): (m_adr[m]=v ∧ 0≤v<V => vm[m]=mm[m] ∧ loc=in) ∧
      (page_inuse[v] ∧ loc=in => ex(vm[v]) ∧ vm[v]=mm[m]) ∧
      (¬page_inuse[v] ∧ loc=out => ex(sm[sm_adr],vm[m])
                        ∧ vm[m] = sm[sm_adr]) ∧
      (¬page_inuse[v] ∧ loc=all_zero => ex(vm[m])
                                        ∧ vm[m] = 0) ∧
      (¬page_inuse[v] ∧ loc=in <=> IAD(v) ∧ ex(sm_adr)
                                    ∧ vm[v]=mm[mm_adr])
```

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

```
Proof of initialization

INPUT ASSERTION: ∀v(0 ≤ v < V) vm[v] = 0 ∧ ex(vm[m])

OUTPUT ASSERTION: I

for m := 0  to  M-1 do
    m_adr[m] := V;
{∀m(0 ≤ m < M): m_adr[m] = V}
/follows from the assignment axiom and the for axiom/
=> {∀m(0 ≤ m < M): m_adr[m] = v ∧ 0 ≤ v < V => vm[v] =
      mm[m]}
   /follows directly since ∀m(0≤m<M): m_adr[m] = V ∧
      V ∉ [0..V-1]/
for v := 0 to V-1 do
   begin
      virtual_map[v].loc := all_zero;
      {virtual_map[v].loc = all_zero}
      page_inuse[v] := false;
      {¬page_inuse[v]}
   end;
{∀v(0 ≤ v < V) virtual_map[v].loc = all_zero ∧
   ¬page_inuse[v]}
/follows from the for axiom/
{virtual_map[v].loc = all_zero => vm[v] = 0}
/follows from the input assertion/
{I}
/follows from the above assertions/

End of proof of initialization
```

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

Proof of the procedure lock_for_input

```
procedure lock_for_input(v: vpf; var s: smpf; var st:
                         (in,out,all_zero));

INPUT ASSERTION: I

OUTPUT ASSERTION: I ∧(st = in => G(v))
                    ∧ (st = out => (sm[s] = vm[v] ∧
                       ex(sm[s], vm[v])))
                    ∧ (st = all_zero => (vm[v] = 0 ∧
                       ex(vm[v])))

begin
   if page_inuse[v] then page_nonbusy[v].wait;
   {I ∧ ¬page_inuse[v]}
   /follows from the wait axiom/
   with virtual_map[v] do
      begin
         st := loc;
         {I ∧¬page_inuse[v] ∧ st = virtual_map[v].loc}
         /follows from the with axiom and the assignment
          axiom/
           case loc of
              in: {I ∧¬page_inuse[v] ∧ st = in ∧
                   virtual_map[v].loc = st ∧ G(v)}
                   page_nonbusy[v].signal;
                   {I ∧ st = in ∧ G(v)}
                   /G(v) follows from signal axiom/
```

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

```
        out: {I ∧¬page_inuse[v] ∧ st = out ∧
              virtual_map[v].loc = st
              ∧ vm[v] = sm[s]}
             /follows from the case axiom/
           begin
              s := sm_adr;
              {s = sm_adr ∧ I ∧¬page_inuse[v] ∧
               st = virtual_map[v].loc ∧ st = out}
              page_inuse[v] := true;
              {st = out ∧ I ∧ page_inuse[v] ∧
               sm[s] = vm[v] ∧ ex(vm[v],sm[s])}
              /sm[s] = vm[s] follows from s = sm_adr
               ∧ ex(vm[v],sm[s]0 follows from the
               input assertion I∧¬page_inuse[v] ∧
               loc = out and the output assertion
               page_inuse[v] that is the result of
               the assignment operation/
           end;
           {I ∧ st = out ∧ ex(vm[v],sm[s])}
      all_zero: {I ∧ ¬page_inuse[v] ∧ st =
                 virtual_map[v].loc ∧ st =
                 all_zero ∧ vm[v] = 0}
                page_inuse[v] := true;
                {I ∧ st = all_zero ∧ vm[v] = 0
                 ∧ ex(vm[v] ∧ page_inuse[v]}
                /follows from the assignment axiom
                 and the input assertion that
                 states the monitor had ex(vm[v])
                 and after the assignment it does
                 not have ex(vm[v])/
      end;
   end;
end lock_for_input;
   {I ∧ (st = in => G(v)) ∧
        (st = out => (sm[s]=vm[m] ∧ ex(vm[v],sm[s]))) ∧
        (st = all_zero => (vm[v] = 0 ∧ ex(vm[v]))}

End of proof of the procedure lock_for_input
```

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

Proof of procedure bringin

```
procedure bringin (v: vpf; m: mmpf; s: sspf);
INPUT ASSERTION: I ∧ page_inuse[v] ∧ vm[v] = mm[m] =
                 sm[s] ∧ virtual_map[v].loc ≠ in ∧
                 ex(vm[v],mm[m],sm[s])
OUTPUT ASSERTION:  I ∧ G(v)
begin
   with virtual_map[v] do
      case loc of
          in: {I ∧ page_inuse[v] ∧ vm[v] = mm[m] = sm[s] ∧
               virtual_map[v].loc ≠ in ∧ virtual_map[v].loc
               = in ∧ ex(vm[v],mm[m],sm[s])}
              /follows from the input assertion and the with
               and case axioms; simplifies to false and is
               an error condition/
               ERROR;
               {I ∧ G(v)}
               /Note that false can imply anything/
         out: {I ∧ page_inuse[v] ∧ vm[v] = mm[m] = sm[s] ∧
               virtual_map[v].loc = out ∧ ex(vm[v],mm[m],
               sm[s])}
              /from the input assertion and the with and
               case axioms/
```

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

```
begin
    mm_adr := m;
    {I ∧ page_inuse[v] ∧ vm[v] = mm[m] = sm[s] ∧
     virtual_map[v].loc = out ∧ virtual_map[v].
     mm_adr = m ∧ ex(vm[m],mm[m],sm[s])}
    /from the input assertion, the assignment
     axiom and the with and case axioms/

    loc := in;
    {I ∧ page_inuse[v] ∧ vm[v] = mm[m] =
     sm[s] ∧ ex(vm[m],mm[m],sm[s]) ∧
     virtual_map[v].loc = in}

    m_adr[m] := v;
    {I ∧ page_inuse[v] ∧ virtual_map[v].loc =
     in ∧ m_adr[m] = v ∧ ex(vm[v], mm[m],sm[s])}

    address_mapper.set_map(v,m);
    {I ∧ page_inuse[v] ∧ virtual_map[v].loc =
     in ∧ m_adr[m] = v ∧ ex(sm[s] ∧ IAD(v)}
    /from the proof of address_mapper.set_map/

    page_inuse[v] = false;
    {I ∧ page_inuse[v] = false ∧ IAD(v)}
    /from the assignment axiom/

    page_nonbusy[v].signal;
    {I ∧ G(v)}
    /from the signal axiom/
end;
    {I ∧ G(v)}
```

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

```
        all_zero: {I ∧ page_inuse[v] ∧ vm[v] = mm[m] =
                   sm[s] ∧ ex(vm[m],mm[m],sm[s]) ∧
                   virtual_map[v].loc = all_zero}

            begin
                mm_adr := m;
                sm_adr := s;
                loc := in;
                m_adr[m] := v;
                page_inuse[v] := false;
                address_mapper.set_map(v,m);
                page_nonbusy[v].signal;
            end;
            {I ∧ G(v)}
            /proof similar to the case when
             virtual_map[v].loc = out/

    end;
end bringin;
  {I ∧ G(v)}
```

End of proof of the procedure bringin

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

```
Proof of the procedure lock_for_output

procedure lock_for_output(m:mmpf; var s: smpf;
                          var output: Boolean);

INPUT ASSERTION: I

OUTPUT ASSERTION: I ∧ (output => (ex(mm[m],sm[s])))
                    ∧ (¬output => F(m))
var v: 0..V;
begin
   output := false;
   {I ∧ ¬output}
   /from the assignment axiom/

   v := m_adr[m];
   {I ∧ ¬output ∧ v = m_adr[m]}
   if (0≤v ∧ v<V) then
   {I ∧ ¬output ∧ v = m_adr[m] ∧ 0≤v<V}
   /from the if axiom/

      begin
         if page_inuse[v] then ERROR;
         {I ∧ ¬output ∧ v = m_adr[m] ∧ 0≤v<V ∧
          ¬page_inuse[v]}
         /follows from the invariant and the fact that
          0≤v<V/

         address_mapper.set_map(V,m);
         {I ∧ ¬output ∧ ex(mm[m],vm[v]) ∧¬page_inuse[v]}
         /follows from the proof of address_mapper.set_map/

         output := true;
         {I ∧ output ∧ ex(mm[m]) ∧ page_inuse[v]}

         s := virtual_map[m].sm_adr;
         {I ∧ output ∧ ex(mm[m],sm[s])}
      end;
         {I ∧ (output => ex(mm[m],sm[s])}
         /by if axiom/
end lock_for_output;
   {I ∧ (output => ex(mm[m],sm[s],vm[m]) ∧ ¬output=>F(m)}

End of proof of the procedure lock_for_output
```

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

Proof of the procedure throwout

```
procedure throwout (m: mmpf; s: smpf);

INPUT ASSERTION: I ∧ page_inuse[m_adr[m]] ∧ mm[m] = sm[s]
                 ∧ ex(mm[m],sm[s]) ∧ 0 ≤ m_adr[m] < V
                 ∧ virtual_map[m_adr[m]].sm_adr = s

OUTPUT ASSERTION: I ∧  ex(mm[m],sm[s]) ∧ F(m)

var v:0 .. V-1;

begin
   v := m_adr[m];
   {I ∧ page_inuse[v] ∧ 0≤v<V ∧ virtual_map[v].sm_adr = s
    ∧ ex(mm[m],sm[s]) ∧ vm[v] = sm[s]}
   /by the use of the assignment axiom and the invariant I/

   m_adr[m] := V;
   {I ∧ page_inuse[v] ∧ 0≤v<V ∧ virtual_map[v].sm_adr = s
    ∧ ex(mm[m],sm[s] ∧ vm[v] = sm[s] ∧ m_adr[m] = V}

   virtual_map[v].loc := out;
   {I ∧ page_inuse[v] ∧ 0≤v<V ∧ virtual_map[v].sm_adr = s
    ∧ ex(mm[m],sm[s]) ∧ vm[v] = sm[s] ∧ m_adr[m] = V
    ∧ virtual_map[v].loc = out}
   /by assignment axiom/
```

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

```
    mm_release(m);
    {I ∧ page_inuse[v] ∧ 0≤v<V ∧ virtual_map[v].sm_adr = s
     ∧ ex(sm[s]) ∧ vm[m] = sm[s] ∧ m_adr = V
     ∧ virtual_map[v].loc = out ∧ F(m) ∧ ¬ex(mm[m])}
    /by the proof of mm_release(m)/

    page_inuse[v] := false;
    {I ∧ ¬(ex(mm[m],sm[s])) ∧ F(m)
     ∧ ¬page_inuse[v]}

    page_nonbusy[v].signal;
    {I ∧ ¬ex(mm[m],sm[s]) ∧ F(m)}
    /by the signal axiom/
end throwout;
    {I ∧ ¬ex(mm[m],sm[s]) ∧ F(m)}
```

End of proof of the procedure throwout

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

Proof of procedure clear_page

```
procedure clear_page(v: vpf);
INPUT ASSERTION: I
OUTPUT ASSERTION: I ∧ vm[v] = 0
begin
   if page_inuse[v] then page_nonbusy[v],wait;
   {I ∧ ¬page_inuse[v]}
   with virtual_map[v] do
      case loc of
          in: {I ∧ ¬page_inuse[v] ∧ virtual_map[v].loc =
               in}
            begin
               address_mapper.set_map(v,mm_adr);
               {I ∧ ¬page_inuse[v] ∧ virtual_map[v].loc =
                 in ∧ ex(mm[m m_adr], vm[m],sm[sm_adr])}
               /follows from the proof of
                address_mapper.set_map/
               m_adr[mm_adr] := V;
               {I ∧ ¬page_inuse[v] ∧ virtual_map[v].
                loc = in ∧ ex(mm[mm_adr],vm[m],sm[sm_adr])
                ∧ m_adr[mm_adr] = V}
               loc := all_zero;
               {I ∧ ¬page_inuse[v] ∧ virtual_map[v].
                 loc = all_zero ∧ ex(mm[mm_adr],vm[v],
                 sm[sm_adr])}
```

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

```
                mm_release(mm_adr);
                {I ∧ ¬page_inuse[v] ∧ vm[v] = 0
                  ∧ ex(vm[v],sm[sm_adr])}
                sm_release(sm_adr);
                {I ∧ ¬page_inuse[v] ∧ vm[v] = 0
                 ∧ ex(vm[v])}
                => {I ∧ ¬page_inuse[v] ∧ vm[v] = 0}
             end;
          out: {I ∧ ¬page_inuse[v] ∧ virtual_map[v].loc =
                out ∧ ex(sm[sm_adr],vm[v])}
             begin
                loc := all_zero;
                {I ∧ ¬page_inuse[v] ∧ ex(sm[sm_adr],
                 vm[v]) ∧ vm[v] = 0}
                sm_release(sm_adr);
                {I ∧ ¬page_inuse[v] ∧ vm[v] = 0}
             end;
          all_zero: {I ∧ ¬page_inuse[v] ∧ vm[v] = 0}
                    do nothing, possible error;
                    {I ∧ ¬page_inuse[v] ∧ vm[v] = 0}
       end;
          {I ∧ ¬page_inuse[v] ∧ vm[v] = 0}
          page_nonbusy[v].signal;
          {I ∧ vm[v] = 0}
end clear_page;
End of proof of the procedure clear_page
End of proof of monitor page_fault_handler
```

Figure C.1 (continued)

Partial Verification of the Simple Memory Manager

Theorem C1:

The process automatic discard ensures that every main memory page frame is in m_pool at least once every N units of time, where N is the maximum time that may be needed to complete one cycle of the repeat statement.

Proof:

The proof follows from the proof of the for statement given below:

```
for m := 0 to M-1 do
   begin
      wait_for_some_time;
      page_fault_handler.lock_for_output(m,s,p_output);
      {(¬p_output => F(m)) ∧ (p_output=>ex(mm[m],sm[s])}
      if p_output then
      {ex(mm[m],sm[s] ∧ mm[m] = mm0[m]}
         begin
            output(m,s);
            {ex(mm[m],sm[s]) ∧ mm[m] = sm[s] = mm0[m]}
            page_fault_handler.throwout(m,s);
            {F(m)}
         end;
            {F(m)}
   end;
   {∀m(0≤m<M):{F(m)}
```

Corollary C1

The process automatic_discarder ensures in cyclic order that every main_memory_page frame is in the m_pool.

Proof:

From the proof of the for statement given above, note that the proof ensures F(m) for m = 0 to M-1 in order and the repeat statement ensures that after making certain that F(M-1) is true, the process starts all over again from m = 0.

C2 PROOF OF TERMINATION AND ABSENCE OF DEADLOCKS

In this section we will prove that all the procedures of the memory manager terminate and that the system is deadlock-free. In section C2.1 we will give the proofs of termination assuming there are no deadlocks and in section C2.2 we will give the proof that the system is deadlock-free.

C2.1. Proof of termination

There are no loops in any program except the procedures fetch, assign and the process automatic discard. The process automatic discard is designed not to terminate. So we need to prove only the termination of the procedures fetch and assign. The proofs are similar so we will only prove the termination of the procedure fetch.

The proof rests on the fact that every time we go around the loop H(v) increases, i.e. the page in question

is brought into the main memory at least once. Now if we assume that the length of time a page resides in the main memory is a random variable τ with mean $\bar{\tau}$, and the time between attempts by a process on the page is also a random variable T with a mean $\bar{T}$, then clearly if $\bar{T} < \bar{\tau}$, the probability that the process will access the page and terminate the loop tends towards 1 as the number of attempts tends towards ∞. T is actually composed of two components T_1 and T_2, where T_1 is the time interval between the time instant when the page_fault was detected and the time instant at which the presence of the page was detected by the process. T_2 is the time interval between the latter instant and the next attempt to access the page. We are interested only in the random variables τ and T_2 and if $\bar{T}_2 \leq \bar{\tau}$ the probability of terminating the loop $\rightarrow 1$ as $H(v) \rightarrow \infty$. (Note T_2 and τ are independent.) If $\max(\tau)$ is less than $\min(T_2)$ then the loop will never terminate, if there is a single process (or unshared pages). However, if there is more than one process, then the page may be brought into the main memory more than once during T_2 and the loop may terminate. If $\max(\tau) > \min(T_2)$ and $\bar{T}_2 > \bar{\tau}$ then also the loop can terminate, even for a single process because probability $(T_2(t) < \tau(t)) > 0$, where t is

the time instant the page was brought into the main memory and $T_2(t)$ and $\tau(t)$ correspond to T_2 and τ with respect to t, and as $t \rightarrow \infty (H(v) \rightarrow \infty)$. Thus the probability that there exists a $t_1 < t$ such that $T_2(t_1) < \tau(t_1)$ tends to 1.

C2.2 Proof of absence of deadlocks

To prove the system to be deadlock-free, we must prove, given that the only outside access to the simple memory manager is through the procedures clear_page, fetch and assign, that for every wait operation on a condition variable there will eventually be a signal operation executed on the same condition variable.

The proof of this result is accomplished in two parts, first we have to prove that all monitors in the system satisfy the following invariant relationship for all condition variables, c, in the monitor:

$$C \Rightarrow (c.queue = empty)$$

where C is the boolean expression involving the data variables of the monitor associated with the condition c and c.queue is the queue of processes waiting on the condition. If this condition is satisfied by the monitor then the proof that the system is deadlock free reduces to proving that whenever the boolean expression C is false

a signal operation on the condition variable c will be eventually executed. This constitutes the second part of the proof of deadlock-free property. We will be using the proof rules given in section 4.2. The deadlock invariant for the monitors is verified in figure C.2.

Proof of the monitor address_mapper:

The monitor contains no condition variables and hence is proved to satisfy the relation vacuously.

Proof of the monitor mm_alloc;

The monitor contains a single condition variable m_nonempty. The boolean condition M_NONEMPTY associated with it is m_pool ≠ empty. The

$$D_{m_nonempty} \text{ is: } |m_pool| = 1 \text{ or } m_pool = [m]$$

thus the invariant relation I_d is

m_pool ≠ empty => (m_nonempty.queue = empty).

Note that initially all condition variable queues are empty, thus {m_nonempty.queue = empty}. After initialization, m_pool := [0..M-1]; {m_pool ≠ empty} hence after initialization the relation Id is satisfied.

Now we have to prove that

$$\{I_d\}\ mm_acquire(m)\ \{I_d\}$$

and

$$\{I_d\}\ mm_release(m)\ \{I_d\}.$$

Figure C2.2

Verification of Deadlock Invariants

Proof of the procedure mm_acquire

```
procedure mm_acquire (var m: mmpf);
   {I_d}
   begin
      if m_pool = empty then {m_pool = empty}
                        m_nonempty.wait;
      {|m_pool| = 1 ∧ m_pool ≠ empty}
      m := anyoneof(m_pool);
      {m_pool = [m]}
      m_pool := m_pool-[m];
      {m_pool = empty}
   end mm_acquire
      {I_d} since m_pool = empty => (m_pool≠empty =>
           m_nonempty.queue = empty)
```

End of proof of the procedure mm_acquire.

Proof of the procedure mm_release

```
procedure mm_release(m:mmpf);
{m_pool = empty => m_nonempty.queue = empty}
begin
   m_pool = m_pool + [m];
   {m_pool ≠ empty ∧ (m_nonempty.queue ≠ empty =>
                      m_pool = [m])}
   m_nonempty.signal;
   {m_pool = empty => m-nonempty.queue = empty}
end mm_release;
```

End of proof of the procedure mm_release

Figure C.2 (continued)

Verification of Deadlock Invariants

Proof of the monitor page_fault_handler

The monitor contains an array of condition variables, page_nonbusy, the range of the index being 0..V-1. The condition PAGE_NONBUSY[v] associated with the condition variable page_nonbusy[v], for $0 \leq v < V$, is $\neg$page_inuse[v]. The condition $D_{page_nonbusy[v]}$ is $I_d(v) \wedge \neg$page_inuse[v] where $I_d(v)$ is the invariant I_d (given below) which is satisfied for all x, $0 \leq x < V$ except x = v.

$$I_d = \forall v(0 \leq v < V): (\neg \text{page_inuse}[v] \Rightarrow \text{page_nonbusy}[v].\text{queue} = \text{empty})$$

Observe that when the procedures are operating on the variables for one value of v, the variables for all other values of v (in the range $0 \leq v < V$) are unaffected. Thus we can prove the monitor by iteration on v for the range $0 \leq v < V$. Note that initially

$$\{\forall v: \text{page_nonbusy}[v].\text{queue} = \text{empty}\}_{(0 \leq v < V)}$$

and after initialization

```
for v := 0 to V-1 do
    page_inuse[v] := false;
```

$$\{\forall v : \neg \text{page_inuse}[v]\}_{(0 \leq v < V)}$$

Hence after initialization I_d is satisfied for all v such that $0 \leq v < V$.

Note that operations on variables other than page_inuse and page_nonbusy do not affect the assertions for proving absence of deadlocks. So in the following proof we have omitted these statements.

Figure C.2 (continued)

Verification of Deadlock Invariants

Proof of the procedure lock_for_input

```
procedure lock_for_input (v: vpf; var s:sspf;
                          var st: (in,out,all_zero);
{I_d}
begin
   if page_inuse[v] then page_nonbusy[v].wait;
   {¬page_inuse[v] ∧ I_d(v)}
   with virtual_map[v] do
      begin
         case loc of
            in: {I_d(v) ∧ ¬page_inuse[v]}
                page_nonbusy[v].signal;
                {I_d ∧ ¬page_inuse[v]}
           out: {I_d(v) ∧ ¬page_inuse[v]}
                begin
                   page_inuse[v] := true;
                   {I_d ∧ page_inuse[v]}
                end;
        all_zero: {I_d(v) ∧ ¬page_inuse[v]}
                  page_inuse[v] := true;
                  {I_d ∧ page_inuse[v]}
      end;
          {I_d}
   end;
end lock_for_input; {I_d}
```

End of proof of the procedure lock_for_input

Figure C.2 (continued)

Verification of Deadlock Invariants

Proof of procedure bringin

```
procedure bringin(v: vpf; m: mmpf; s: sspf);
{I_d ∧ page_inuse[v]}
begin
   with virtual_map[v] do
      case loc of
         in: ERROR;
             {I_d ∧ page_inuse[v]}
        out: {I_d ∧ page_inuse[v]}
             begin
                page_inuse[v] := false;
                {I_d(v) ∧ ¬page_inuse[v]}
                page_nonbusy[v].signal;
                {I_d}
             end
                {I_d}
      all_zero: {I_d}
             begin
                page_inuse[v] := false;
                {I_d(v) ∧ ¬page_inuse[v]}
                page_nonbusy[v].signal;
                {I_d}
             end;
      end; {I_d}
end bringin; {I_d}
```

End of proof of the procedure bringin

Figure C.2 (continued)

Verification of Deadlock Invariants

Proof of procedure clear_page

```
procedure clear_page(v: vpf);
{I_d}
begin
   if page_inuse[v] then page_nonbusy.wait;
   {I_d(v) ∧ ¬page_inuse}
   page_nonbusy[v].signal;
   {I_d}
end clear_page;
   {I_d}
```

End of proof of clear_page.

Figure C.2 (continued)

Verification of Deadlock Invariants

Proof of the procedure lock_for_output

The procedure does not have any wait or signal statements. Hence the invariant Id is satisfied at output, since it is satisfied at input.

End of proof of the procedure lock_for_output.

```
procedure throwout(m: mmpf);
{I_d ^ page_inuse[m_adr[m]]}
var v: 0..V-1
begin
   v := m_adr[m];
   {I_d ^ page_inuse[v]}
   page_inuse[v] := false;
   {I_d(v) ^ ¬page_inuse[v]}
   page_nonbusy[v].signal;
   {I_d}
end throwout;
   {I_d}
end of proof of throwout
```

End of proof of monitor page_fault_handler

Figure C.2 (continued)

Verification of Deadlock Invariants

Having proved that when the condition C associated with a condition variable c is satisfied, no process will be waiting (blocked), we now prove that whenever the condition C is not satisfied a signal operation, after satisfying the condition, on the condition variable c will be executed. Note that this result is stronger than what we need; we need only prove that whenever C is not satisfied and a process is waiting on c, a signal operation on c will be executed.

To prove this result, we make the following observations:

1. No process can be waiting on more than one condition.
2. If we consider each condition to be a resource, since it controls access to a resource, and if we prove that every process that acquires a resource eventually releases it, then we have proved that no process will wait on a condition forever (assuming "fair" resource allocation policy, like the FIFO implemented in our system). This is so because, the resources controlled by the condition variable initially belong to the monitor containing the condition variable and we can verify that whenever a resource is released (condition satisfied) a signal operation on the associated

condition variable is executed (verified in C2.2).

We have only three monitors, mm_alloc, sm_alloc, and page_fault_handler that contain condition variables.

The resource controlled by the monitor mm_alloc's condition variable m-nonempty is a main memory page_frame. If m is a main memory page_frame then mm[m], the contents of the main memory page_frame is the resource represented in our assertions. An inspection of the proofs in section C1 shows that the only means of acquiring this resource from the monitor is by the use of the procedure mm_acquire. The procedure page_fault is the only procedure that invokes the procedure mm_acquire and once it acquires this resource it always releases it to the monitor page_fault_handler (assuming termination of the procedure input). The monitor page_fault_handler in turn releases the resource to the monitor address_mapper and then reacquires it from the monitor address_mapper, when the procedure lock_for_input is invoked. Note that the process automatic_discard ensures that lock_for_input is invoked for every main memory page_frame m. Note also that the monitor address_mapper contains no condition variables and so the resource can be always reacquired.

Further when the procedure lock_for_input is invoked and the resource is currently owned by the monitor address_-mapper, the process invoking the procedure lock_for_input will not be blocked since, as per the invariant of the monitor page_fault_handler, the boolean variable page_-inuse[v] corresponding to the mmpf will be false. The monitor page_fault_handler, then releases the resource to the process automatic discard. The process automatic discard always returns the resource to the monitor page_-fault_handler through the procedure throwout. The procedure throwout always returns the resource to the monitor mm_alloc. Thus there can be no deadlock due to the resource mm[m] for $0 \leq m < M-1$.

The resource sm[s] controlled by the monitor sm_alloc is acquired only by a process through the procedure page_-fault. The process always (note that there will be no deadlock or permanent blocking in acquiring mm[m] after acquiring sm[s], as proved earlier) releases the resource to the monitor page_fault_handler. The resource is never granted to a process by the monitor page_fault_handler. It can only be released to the monitor sm_alloc through the invokation of the procedure clear_page. Now we cannot prove that the procedure clear_page will always be invoked,

thus it is possible for a process to be permanently blocked on this resource. This can happen, if all the secondary memory space is allocated and we want to access virtual pages that are currently all_zero and no virtual_pages are cleared. This is a case of requiring more resources than are available. A possible solution is for the system to detect all_zero pages that are not cleared, but are occupying secondary memory space and clear these pages. If there are no such pages, then the system can ask the users to clear some pages or destroy some pages itself based on some policy (dangerous).

The monitor page_fault_handler controls the resource vm[v] through the condition ¬page_inuse[v] and the condition variable page_nonbusy[v]. It grants the resource to the monitor address_mapper when procedure bringin is invoked but always reacquires it when the procedure clear_page or the procedure lock_for_output is invoked. When the resource belongs to the monitor address_mapper, then one of these procedures can always be invoked. The input assertions for the procedures bringin and throwout are not satisfied and the procedure lock_for_input does not need the resource.

When the process automatic discard acquires the resource through the procedure lock_for_output, it always releases it through the procedure throwout.

When a process acquires the resource through the procedure lock_for_input, invoked by the procedure page_fault, it always releases the resource if the page was on secondary memory (st = out), if the page was an all_zero page (st = all_zero) then the resource is released only if the process (procedure page_fault) can acquire a smpf.

Thus the deadlock_free nature of the system is guaranteed only if there is sufficient secondary memory. However, in case there is insufficient main_memory the process can only block those processes that require access to the same virtual_page, or those requiring access to other all_zero virtual_pages.

BIBLIOGRAPHY

[Bredt and Saxena 1974]

T. H. Bredt and A. R. Saxena, "Hierarchical Design Methods for Operating Systems." Digest IEEE Computer Society International Conference (September 1974), pp. 153-156.

[Brinch Hansen 1973]

P. Brinch Hansen, Operating System Principles. Prentice Hall, Englewood Cliffs, New Jersey (1973).

[Brinch Hansen 1975]

P. Brinch Hansen, "The Purpose of Concurrent Pascal." Proceedings 1975 International Conference on Reliable Software. Los Angeles (21-23 April 1975), pp. 305-309.

[Coffman et al. 1971]

E. G. Coffman, Jr., M. J. Elphick, and A. Shoshani, "System Deadlocks". ACM Computing Surveys 3,2(June 1971), pp. 67-78.

[Coffman and Denning 1973]

E. G. Coffman, Jr., and P. J. Denning, Operating Systems Theory. Prentice-Hall, Englewood Cliffs, New Jersey (1973).

[Dahl, Dijkstra, and Hoare 1972]

O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Structured Programming. Academic Press, London and New York, (1972).

[Dahl and Hoare 1972]

O. J. Dahl, and C. A. R. Hoare, "Hierarchical Program Structures". In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Structured Programming. Academic Press, London and New York (1972).

[Deutsh 1973]

L. P. Deutsh, "An Interactive Program Verifier". Ph.D. Thesis, Computer Science Department, University of California, Berkeley (June 1973).

[Dijkstra 1968a]

E. W. Dijkstra, "Complexity controlled by hierarchical ordering of function and variability". In B. Randell and P. Naur (ed.), Report on a Conference on Software Engineering. NATO 1968.

[Dijkstra 1968b]

E. W. Dijkstra, "The structure of the 'THE' multi-programming system". CACM 11,5 (May 1968), pp. 341-346.

[Dijkstra 1968c]

E. W. Dijkstra, "Go to statement considered harmful". CACM 11,3 (March 1968), pp. 147-148,538,541.

[Floyd 1967]

R. W. Floyd, "Assigning Meanings to Programs". Proceedings of a Symposium on Applied Mathematics, American Mathematical Society 19(1967), pp. 19-32.

[Hoare 1969]

C. A. R. Hoare, "An Axiomatic Basis for Computer Programming". CACM 12,10 (October 1969), pp. 576-580,583.

[Hoare 1971a]

C. A. R. Hoare, "Procedures and Parameters: An Axiomatic Approach". Symposium on Semantics of Algorithmic Languages, Springer-Verlag, Berlin, Heidelberg, New York (1971), pp. 102-116.

[Hoare 1971b]

C. A. R. Hoare, "Proof of a Program: FIND". CACM 14,1 (January 1971), pp. 39-45.

[Hoare 1973]

C. A. R. Hoare, "A Structured paging system". Computer Journal 16,3 (August 1973), pp. 209-215.

[Hoare 1974]

C. A. R. Hoare, "Monitors: an operating system structuring concept". CACM 17,10 (October 1974), pp. 549-557.

[Hoare and Wirth 1972]

C. A. R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL". Eidgenossische Technische Hochschule, Zurich, Switzerland (November 1972).

[Holt 1971]

R. C. Holt, "Comments on Prevention of System Deadlocks". CACM 14,1 (January 1971), pp. 36-38.

[Howard 1975]

J. H. Howard, "Proving Monitors". Fifth Symposium on Operating Systems Principles. Austin, Texas (19-21 November 1975). [To appear in CACM 19,4 (April 1976)].

[IBM]

IBM System 360 Principles of Operation.

[Igarashi et al. 1973]

S. Igarashi, R. L. London, and D. C. Luckham, "Automatic Verification of Programs I: A Logical Basis and Implementation". Computer Science Department Report CS365, AIM 200, Stanford University (May 1973).

[Katzan 1973]

H. Katzan, Jr., Operating Systems: A Pragmatic Approach. Van Nostrand Reinhold, New York (1973).

[King 1969]

J. C. King, "A Program Verifier". Ph.D. Thesis, Carnegie-Mellon University (1969).

[Knuth 1974]

D. E. Knuth, "Structured Programming with go to Statements". ACM Computing Surveys 6,4 (December 1974), pp. 261-301.

[Liskov 1972]

B. H. Liskov, "The design of the Venus operating system". CACM 15,3 (March 1972), pp. 142-149.

[London 1970]

R. L. London, "Certification of Algorithm 245[M1] Treesort 3: Proof of Algorithms -- A New Kind of Certification. CACM 13,6 (June 1970), pp. 371-373.

[Manna 1974]

Z. Manna, Mathematical Theory of Computation. McGraw-Hill, New York, St. Louis, San Francisco (1974).

[Neumann *et al.* 1975]

P. G. Neumann, L. Robinson, K. N. Levitt, R. S. Boyer, and A. R. Saxena, A Provably Secure Operating System. Final Report, SRI Project 2581, Stanford Research Institute (June 1975).

[Organick 1972]

E. I. Organick, The Multics System: An Examination of its Structure. M. I. T. Press, Cambridge, Massachusetts (1972).

[Owicki and Gries 1975]

S. Owicki and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach". Fifth Symposium on Operating Systems Principles. Austin, Texas (19-21 November 1975). [To appear in CACM 19, 4 (April 1976)].

[Parnas 1972]

D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules". CACM 15,12 (December 1972), pp. 1053-1058.

[Parnas 1974]

D. L. Parnas, "On a 'Buzzward': Hierarchical Structure". Proceedings of the IFIP Congress 1974. Stockholm, Sweden (1974), pp. 336-339.

[Saxena 1975]

A. R. Saxena, "An Efficient Implementation of Monitors and Condition Variables". Digital Systems Laboratory, Technical Note 72, Stanford University (August 1975).

[Saxena and Bredt 1975]

A. R. Saxena and T. H. Bredt, "A Structured Specification of a Hierarchical Operating System". Proceedings 1975 International Conference on Reliable Software, Los Angeles (21-23 April 1975), pp. 310-318.

[Wirth 1972]

N. Wirth, "The Programming Language PASCAL". Revised Report, Eidgenossische Technische Hochschule, Zurich, Switzerland (November 1972).

[Wirth 1974]

N. Wirth, "On the Composition of Well-Structured Programs". ACM Computing Surveys 6,4 (December 1974), pp. 247-259.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: DSL T.R. #107, SEL-76-011

2. GOVT ACCESSION NO.: TR-107, SU-SEL-76-011

3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): A Verified Specification of a Hierarchical Operating System

5. TYPE OF REPORT & PERIOD COVERED: Technical Report

6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): Ashok R. Saxena

8. CONTRACT OR GRANT NUMBER(s): NSF-GJ41644, JSEP N00014-75-C-0601

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Stanford Electronics Laboratories
Stanford University
Stanford, CA 94305

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 255p.

11. CONTROLLING OFFICE NAME AND ADDRESS:
Sponsored Projects
Stanford University

12. REPORT DATE: January 1976

13. NUMBER OF PAGES: 252

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

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Reproduction in whole or in part is permitted for any purpose of the U. S. Government

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)

Operating systems, Hierarchies, Mutual exclusion, Structured programming, Monitors, Verification, Processor allocation, Virtual memory, PASCAL, Program correctness.

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

This thesis discusses the design, specification, and verification of computer operating systems. The operating system problem considered, the many-process problem, is the design of an operating system that can support a large number of concurrent processes. This design problem is a vehicle to investigate the use of a design methodology, the hierarchical levels of abstraction methodology; the use of structured programming techniques in the specification of the system; and the development of techniques for the verification of concurrent programs, particularly operating system programs.

OVER

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

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

332 400

A solution to the many-process problem is obtained and it is shown that the hierarchical levels of abstraction methodology simplifies the conception of the solution and helps avoid potential deadlocks in the system. A PASCAL specification of the four levels of the system is given demonstrating the usefulness of structured programming techniques for specifying operating system programs. A detailed description of the development of the simple memory manager, a complex and large segment of the system, is given to show the use of step-wise refinement for improving the efficiency of the program and as an aid in understanding its final specification. The specifications for the first two levels: simple scheduler and simple memory manager, are formally verified. The notion of exclusive access of a resource has been formalized and used in the verification of concurrent program. Sufficient conditions for verifying the absence of deadlocks in a system of monitors are also developed.

JSEP REPORTS DISTRIBUTION LIST

	No. of Copies
Department of Defense	
Defense Documentation Center Attn: DDC-TCA (Mrs. V. Caponio) Cameron Station Alexandria, Virginia 22314	12
Asst. Dir., Electronics and Computer Sciences Office of Director of Defense Research and Engineering The Pentagon Washington, D.C. 20315	1
Office of Director of Defense Research and Engineering Information Office Lib. Branch The Pentagon Washington, D.C. 20301	1
ODDR&E Advisory Group on Electron Devices 201 Varick Street New York, New York 10014	1
Chief, R&D Division (340) Defense Communications Agency Washington, D.C. 20301	1
Director, Nat. Security Agency Fort George G. Meade Maryland 20755 Attn: Dr. T. J. Beahn	1
Institute for Defense Analysis Science and Technology Div. 400 Army-Navy Drive Arlington, Virginia 22202	1
Dr. Stickley Defense Advanced Research Projects Agency Attn: Technical Library 1400 Wilson Boulevard Arlington, Virginia 22209	1
Dr. R. Reynolds Defense Advanced Research Projects Agency Attn: Technical Library 1400 Wilson Boulevard Arlington, Virginia 22209	1
Department of the Air Force	
AF/RDPS The Pentagon Washington, D.C. 20330	1
AFSC (LJ/Mr. Irving R. Mirman) Andrews Air Force Base Washington, D.C. 20334	1
Directorate of Electronics and Weapons HQ AFSC/DLC Andrews AFB, Maryland 20334	1
Directorate of Science HQ AFSC/DLS Andrews Air Force Base Washington, D.C. 20334	1
LTC J. W. Gregory AF Member, TAC Air Force Office of Scientific Research Bolling Air Force Base Washington, D.C. 20332	5
Mr. Carl Sletten RADC/ETE Hanscom AFB, Maryland 01731	1
Dr. Richard Picard RADC/ETSL Hanscom AFB, Maryland 01731	1
Mr. Robert Barrett RADC/ETS Hanscom AFB, Maryland 01731	1

	No. of Copies
Dr. John N. Howard AFGL/CA Hanscom AFB, Maryland 01731	1
Dr. Richard B. Mack RADC/ETER Hanscom AFB, Maryland 01731	1
Documents Library (TILD) Rome Air Development Center Griffiss AFB, New York 13441	1
Mr. H. E. Webb, Jr. (ISCP) Rome Air Development Center Griffiss AFB, New York 13441	1
Mr. Murray Kesselman (ISCA) Rome Air Development Center Griffiss AFB, New York 13441	1
Mr. W. Edwards AFAL/TE Wright-Patterson AFB Ohio 45433	1
Mr. R. D. Larson AFAL/DHR Wright-Patterson AFB Ohio 45433	1
Howard H. Steenbergen AFAL/DHE Wright-Patterson AFB Ohio 45433	1
Chief Scientist AFAL/CA Wright-Patterson AFB Ohio 45433	1
HQ ESD (DRI/Stop 22) Hanscom AFB, Maryland 01731	1
Professor R. E. Fontana Head, Dept. of Elec. Eng. AFIT/ENE Wright-Patterson AFB Ohio 45433	1
Mr. John Mottsmith (MCIT) HQ ESD (AFSC) Hanscom AFB, Maryland 01731	1
LTC Richard J. Gowen Professor, Dept. of Elec. Eng. USAF Academy, Colorado 80840	1
AUL/LSE-9663 Maxwell AFB, Alabama 36112	1
AFETR Technical Library P. O. Box 4608, MU 5650 Patrick AFB, Florida 32542	1
ADTC (DLOSL) Eglin AFB, Florida 32542	1
HQ AMD (RDR/Col. Godden) Brooks AFB, Texas 78235	1
USAF European Office of Aerospace Research Technical Information Office Box 14, FPO, New York 09510	1
Dr. Carl E. Baum AFWL (ES) Kirtland AFB, New Mexico 87117	1
ASAFSAM/RAL Brooks AFB, Texas	1
Department of the Army	
HQDA (DAMA-ARZ-A) Washington, D.C. 20310	1
Commander U.S. Army Security Agency Attn: IARD-T Arlington Hall Station Arlington, Virginia 22212	1
Commander U.S. Army Materiel Development and Readiness Command Attn: Tech. Lib. Rm. 7S 35 5001 Eisenhower Avenue Alexandria, Virginia 22333	1

	No. of Copies
Commander U.S. Army Ballistics Research Laboratory Attn: DRXRD-BAD Aberdeen Proving Ground Aberdeen, Maryland 21005	1
Commander Picatinny Arsenal Attn: SMUPA-TS-T-S Dover, New Jersey 07801	1
U.S. Army Research Office Attn: Dr. Hermann Robl P. O. Box 12211 Research Triangle Park North Carolina 27709	1
U.S. Army Research Office Attn: Mr. Richard O. Ulsh P. O. Box 12211 Research Triangle Park North Carolina 27709	1
U.S. Army Research Office Attn: Dr. Jimmie R. Suttle P. O. Box 12211 Research Triangle Park North Carolina 27709	1
U.S. Army Research Office Attn: Dr. Horst Wittmann P. O. Box 12211 Research Triangle Park North Carolina 27709	1
Commander Frankford Arsenal Attn: Mr. G. C. White, Jr. Deputy Director, Pitman-Dunn Laboratory Philadelphia, Pa. 19137	1
Commander U.S. Army Missile Command Attn: Chief, Document Sec. Redstone Arsenal, Al. 35809	1
Commander U.S. Army Missile Command Attn: DRSMI-RR Redstone Arsenal, Al. 35809	1
Commander U.S. Army Materials and Mechanics Research Center Attn: Chief, Materials Sciences Division Watertown, Ma. 02172	1
Commander Harry Diamond Laboratories Attn: Mr. John E. Rosenberg 2800 Powder Mill Road Adelphi, Maryland 20783	1
Commandant U.S. Army Air Defense School Attn: ATSAD-T-CSM Fort Bliss, Texas 79916	1
Commandant U.S. Army Command and General Staff College Attn: Acquisitions, Lib. Div. Fort Leavenworth, Kansas 66027	1
Dr. Hans K. Ziegler Army Member, TAC/JSEP U.S. Army Electronics Command (DRSEL-TL-D) Fort Monmouth, N.J. 07703	1
Mr. J. E. Teti Executive Secretary, TAC/JSEP U.S. Army Electronics Command (DRSEL-TL-DT) Fort Monmouth, N.J. 07703	3
Director Night Vision Laboratory, ECOM Attn: DRSEL-NV-D Fort Belvoir, Virginia 22060	1
Commander/Director Atmospheric Sciences Lab. (ECOM) Attn: DRSEL-BL-DD White Sands Missile Range New Mexico 88002	1

	No. of Copies
Director Electronic Warfare Laboratory (ECOM) Attn: DRSEL-WL-MY White Sands Missile Range New Mexico 88002	1
Commander U.S. Army Armament Command Attn: DRSAR-RD Rock Island, Illinois 61201	1
Director, Division of Neuropsychiatry Walter Reed Army Institute of Research Washington, D.C. 20012	1
Commander USASATCOM Fort Monmouth, N.J. 07703	1
Commander U.S. Army R&D Group (Far East) APO San Francisco, Ca. 96343	1
Commander U.S. Army Communications Command Attn: Director, Advanced Concepts Office Fort Huachuca, Az. 85613	1
Project Manager ARTADS EAI Building West Long Branch, N.J. 07764	1
Commander U.S. Army White Sands Missile Range Attn: STEWS-ID-R White Sands Missile Range New Mexico 88002	1
Director, TRI-TAC Attn: TT-AD (Mrs. Briller) Fort Monmouth, N.J. 07703	1
Commander U.S. Army Communication Command Attn: CC-OPS-PD Fort Huachuca, Az. 85613	1
COL Robert Noce Senior Standardization Representative U.S. Army Standardization Group, Canada Canadian Force Headquarters Ottawa, Ontario, CANADA K1A 0K2	1
Commander U.S. Army Electronics Command Attn: DRSEL-RD-O (Dr. W. S. McAfee)	1
DRSEL-CT-L (Dr. R. Buser)	1
DRSEL-NL-O (Dr. H. S. Bennett)	1
DRSEL-NL-T (Mr. R. Kulinyi)	1
DRSEL-TL-B	1
DRSEL-VL-D	1
DRSEL-WL-D	1
DRSEL-TL-NM (Mr. N. Lipetz)	1
DRSEL-NL-H (Dr. F. Schwering)	1
DRSEL-TL-E (Dr. S. Kronenberg)	1
DRSEL-TL-E (Dr. J. Kohn)	1
DRSEL-TL-I (Dr. C. Thornton)	1
DRSEL-NL-B (Dr. S. Amoroso) Fort Monmouth, N.J. 07703	1
Project Manager Ballistic Missile Defense Program Office Attn: DACS-BMP (Mr. A. Gold) 1300 Wilson Boulevard Washington, D.C. 22209	1

	No. of Copies
Department of the Navy	
Office of Naval Research Electronic and Solid State Sciences Program (Code 427) 800 N. Quincy Arlington, Virginia 22217	1
Office of Naval Research Code 200 Asst. Chief for Technology 800 N. Quincy Arlington, Virginia 22217	1
Office of Naval Research Information Sciences Program (Code 437) 800 N. Quincy Arlington, Virginia 22217	1
Naval Research Laboratory 4555 Overlook Avenue, S.W. Washington, D.C. 20375	
Attn: Code 2627	1
4000	1
4105	1
5000	1
5200	1
5203	1
5210	1
5270	1
5300	1
5400	1
5460	1
5464	1
5500	1
5510	1
6400	1
Director Office of Naval Research Branch Office 536 South Clark Street Chicago, Illinois 60605	1
San Francisco Area Office Office of Naval Research 760 Market Street, Rm. 447 San Francisco, Ca. 94102	1
Dr. A. Laufer Chief Scientist Office of Naval Research Branch Office 1030 East Green Street Pasadena, California 91101	1
Director Office of Naval Research Branch Office 715 Broadway, 5th Floor New York, New York 10003	1
New York Area Office Office of Naval Research 715 Broadway, 5th Floor New York, New York 10003	1
Mr. L. W. Sumney Naval Electronics Systems Command NC #1 2511 Jefferson Davis Highway Arlington, Virginia 20360	1
Mr. R. Fratila Naval Electronics Systems Command NC #1 2511 Jefferson Davis Highway Arlington, Virginia 20360	1
Mr. N. Butler Naval Electronics Systems Command NC #1 2511 Jefferson Davis Highway Arlington, Virginia 20360	1
Dr. H. J. Meuller Naval Air Systems Command JP #1 1411 Jefferson Davis Highway Arlington, Virginia 20360	1
Capt. R. B. Meeks Naval Sea Systems Command NC #3 2531 Jefferson Davis Highway Arlington, Virginia 20362	1

	No. of Copies
Commander Naval Surface Weapons Center Attn: Technical Library Silver Spring, Maryland 29010	1
Naval Surface Weapons Center Attn: Code 212 Silver Spring, Maryland 29010	1
Officer-in-Charge Naval Surface Weapons Center Dahlgren Laboratory Dahlgren, Virginia 22448	1
Naval Air Development Center Attn: Technical Library Johnsville Warminster, Pa. 18974	1
Commander Naval Avionics Facility Indianapolis, Indiana 46241 Attn: D/035 Tech. Library	1
Naval Missile Center Technical Library Code 5632.2 Point Mugu, California 93042	1
Naval Weapons Center Attn: Tech. Lib., Code 533 China Lake, California 93555	1
Naval Weapons Center Attn: Code 6010 China Lake, California 93555	1
Naval Weapons Center Mathematics Division China Lake, California 93555	1
Naval Training Equipment Center Technical Library Orlando, Florida 32813	1
Naval Research Laboratory Underwater Sound Reference Div. Technical Library P. O. Box 8337 Orlando, Florida 32806	1

	No. of Copies
Naval Underwater Sound Center Technical Library New London, Conn. 06320	1
U.S. Naval Oceanographic Off. Library - Code 1600 Washington, D.C. 20373	1
Commandant, Marine Corps Scientific Advisor (Code AX) Washington, D.C. 20380	1
Dr. Gernot M. R. Winkler Director, Time Service U.S. Naval Observatory Massachusetts Avenue at 34th Street N.W. Washington, D.C. 20390	1
Naval Postgraduate School Technical Library Monterey, California 93940	1
Naval Electronics Lab. Center Technical Library San Diego, California 92152	1
Naval Electronics Lab. Center Attn: Code 2600 San Diego, California 92152	1
Naval Electronics Lab. Center San Diego, California 92152	
Attn: Code 2000	1
2200	1
2300	1
3000	1
3200	1
3300	1
4800	1
5000	1
5200	1
5300	1
Naval Undersea Center Technical Library San Diego, California 92152	1

	No. of Copies
Naval Ship Research and Development Center David W. Taylor Code 522.1 Bethesda, Maryland 20084	1
Office of Chief of Naval Operations NAICOM/MIS Planning Branch NOP-916D, Pentagon Washington, D.C. 20350	1

Other Government Agencies

Mr. F. C. Schwenk, RD-T National Aeronautics and Space Administration Washington, D.C. 20546	1
Los Alamos Scientific Lab. Attn: Reports Library P. O. Box 1663 Los Alamos, N.M. 87544	1
M. Zane Thornton Deputy Director, Institute for Computer Sciences and Technology National Bureau of Standards Washington, D.C. 20234	1
Director, Office of Postal Technology (R & D) U.S. Postal Service 11711 Parklawn Drive Rockville, Maryland 20852	1
NASA Lewis Research Center Attn: Library 21000 Brookpark Road Cleveland, Ohio 44135	1
Library - R51 Bureau of Standards Acquisition Boulder, Colorado 80302	1
MIT Lincoln Laboratory Attn: Library A-082 P. O. Box 73 Lexington, Ma. 02173	1
Dr. Jay Harris Program Director, Devices and Waves Program National Science Foundation 1800 G Street Washington, D.C. 20550	1
Dr. Howard W. Etzel Deputy Director, Division of Materials Research National Science Foundation 1800 G Street Washington, D.C. 20550	1
Dr. Dean Mitchell Program Director, Solid-State Physics Div. of Materials Research National Science Foundation 1800 G Street Washington, D.C. 20550	1

Non-Government Agencies

Director Research Lab. of Electronics Massachusetts Inst. of Tech. Cambridge, Ma. 02139	1
Director Microwave Research Institute Polytechnic Inst. of New York Long Island Graduate Center Route 110 Farmingdale, New York 11735	1
Assistant Director Microwave Research Institute Polytechnic Inst. of New York 333 Jay Street Brooklyn, New York 11201	1

	No. of Copies
Director Columbia Radiation Laboratory Department of Physics Columbia University 538 West 120th Street New York, New York 10027	1
Director Coordinated Science Laboratory University of Illinois Urbana, Illinois 61801	1
Director Stanford Electronics Lab. Stanford University Stanford, California 94305	1
Director Microwave Laboratory Stanford University Stanford, California 94305	1
Director Electronics Research Lab. University of California Berkeley, California 94720	1
Director Electronics Sciences Lab. University of Southern California Los Angeles, California 90007	1
Director Electronics Research Center The Univ. of Texas at Austin Engineering-Science Bldg. 112 Austin, Texas 87812	1
Director of Laboratories Division of Engineering and Applied Physics Technical Reports Collection Harvard University Pierce Hall Cambridge, Ma. 02138	1