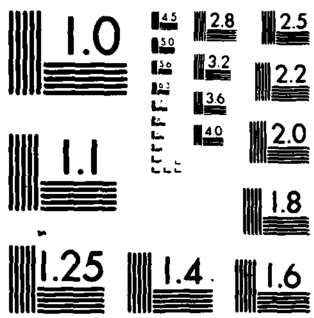
AD-A091 092 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
AN IMPLEMENTATION OF MULTIPROGRAMMING AND PROCESS MANAGEMENT FO--ETC(U)
JUN 80 S L REITZ

UNCLASSIFIED NL

1 of 2


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

AN IMPLEMENTATION OF MULTIPROGRAMMING AND PROCESS MANAGEMENT FOR A SECURITY KERNEL OPERATING SYSTEM

by

Stephen Leslie Reitz

June 1980

Thesis Advisor: R. R. Schell

Approved for public release; distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO. AD-A091092

3. RECIPIENT'S CATALOG NUMBER

9

4. TITLE (and Subtitle)

6

An Implementation of Multiprogramming and Process Management for a Security Kernel Operating System.

5. TYPE OF REPORT & PERIOD COVERED

Master's Thesis, June 1980

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

10

Stephen Leslie Reitz

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Naval Postgraduate School
Monterey, California 93940

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

11. CONTROLLING OFFICE NAME AND ADDRESS

Naval Postgraduate School
Monterey, California 93940

11

12. REPORT DATE

Jun[redacted]80

13. NUMBER OF PAGES

140

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

12 141

15. SECURITY CLASS. (of this report)

UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

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

18. SUPPLEMENTARY NOTES

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

operating systems, distributed computer networks, security kernel, computer security, microprocessors, archival storage

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

This thesis presents an implementation of multiprogramming and process management functions for the security kernel of a distributed multiprocessor system. The implementation is based on a family of operating systems designed to provide controlled access in a microcomputer network to data bases containing multiple levels of sensitive information.

Multiprogramming improves system efficiency and creates a virtual

environment which frees the remainder of the operating system from a dependence on processor configuration. Processor management coordinates the asynchronous interaction of system processes.

This implementation describes a processor multiplexing technique for a distributed kernel and presents a virtual interrupt mechanism. Its structure is loop free to permit future expansion into more complex members of the design family.

Approved for public release; distribution unlimited.

An Implementation of Multiprogramming and Process Management for a Security Kernel Operating System

by

Stephen Leslie Reitz
Lieutenant Commander, United States Navy
BS, Purdue University, 1971

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1980

Author ______________________ Stephen L. Reitz

Approved by: ______________________ Roger R. Schell
Thesis Advisor

______________________ Lyle A. Cox Jr.
Second Reader

____
Chairman, Department of Computer Science

____
Dean of Information and Policy Sciences

ABSTRACT

This thesis presents an implementation of multiprogramming and process management functions for the security kernel of a distributed multiprocessor system. The implementation is based on a family of operating systems designed to provide controlled access in a microcomputer network to data bases containing multiple levels of sensitive information.

Multiprogramming improves system efficiency and creates a virtual environment which frees the remainder of the operating system from a dependence on processor configuration. Processor management coordinates the asynchronous interaction of system processes.

This implementation describes a processor multiplexing technique for a distributed kernel and presents a virtual interrupt mechanism. Its structure is loop free to permit future expansion into more complex members of the design family.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENT

This research is sponsored in part by Office of Naval Research Project number NR 337-005, monitored by Mr. Joel Trimble.

I am indebted to a number of people for the support they have given me in completing this thesis. Lt. Col Roger Schell, my advisor, was a never ending source of new ideas. He provided me with solutions to many seemingly unsolvable problems, and I greatly appreciate the many hours he has spent helping me to clarify my work. Without his able and enthusiastic guidance, this thesis could not have been written.

Mike Williams and Bob McDonnell helped me with many hardware problems that I encountered in getting up and running on an unfamiliar system.

Finally, I would like to thank my wife, Madelyn, and my children, Stephen and Monica for their patience and understanding. They won't have to tip-toe around the house any more.

I. INTRODUCTION

The application of contemporary microprocessor technology to the design of large-scale multiple processor systems offers many potential benefits. The cost of high-power computer systems could be reduced drastically; fault tolerance in critical real-time systems could be improved; and computer services could be applied in areas where their use is not now cost effective. Designing such systems presents many formidable problems that have not been solved by the specialized single processor systems available today.

Specifically, there is an increasing demand for computer systems that provide protected storage and controlled access for sensitive information to be shared among a wide range of users. Data controlled by the Privacy Act, classified Department of Defence (DoD) information, and the transactions of financial institutions are but a few of the areas which require protection for multiple levels of sensitive information. Multiple processor systems which share data are well suited to providing such services - if the data security problem can be solved.

A solution to these problems - a multiprocessor system design with verifiable information security - is offered in a family of secure, distributed multi-microprocessor operating systems designed by O'Connell and Richardson [1]. A subset of this family, the Secure Archival Storage System (SASS) [2,3], has been selected as a testbed for the general design. SASS will provide consolidated file storage for a network of possibly dissimilar "host" computers. The system will provide controlled, shared access to multiple levels of sensitive information (figure 1).

This thesis presents an implementation of a basic monitor for the O'Connell-Richardson family of operating systems. The monitor provides multiprogramming and process management functions specifically addressed to the control of physical processor resources of SASS. Concurrent thesis work [4] is developing a detailed design for a security kernel process, the Memory Manager, which will manage SASS memory resources.

SASS SYSTEM

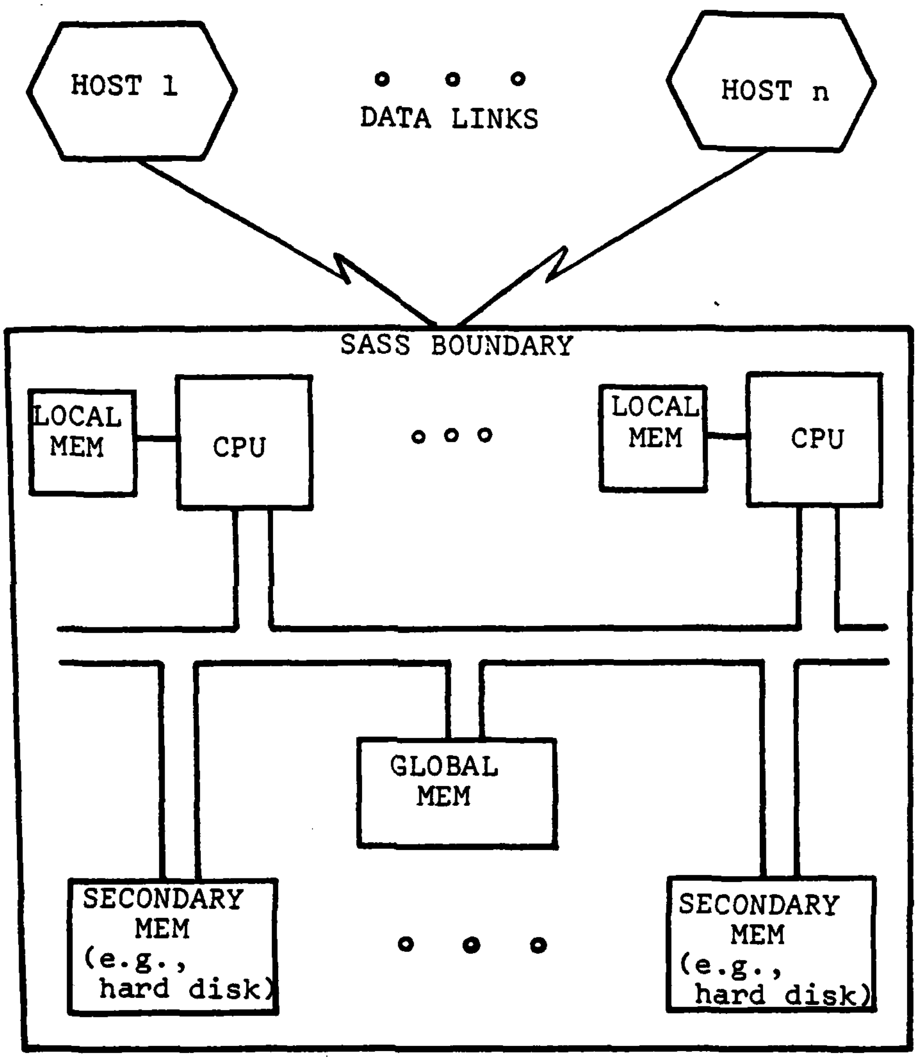


Figure 1

A. BACKGROUND

The general family design is composed of a supervisor and a security kernel. The supervisor provides dynamic linking, a discretionary security policy, demand memory management, and a hierarchical file system in support of the user. The security kernel manages physical resources to provide scheduling, interprocess communication and synchronization, and a non-discretionary security policy. The design is loop-free to permit the implementation of system subsets ranging from a simple monitor to a general purpose computer utility.

SASS is a subset of this system and does not require use of several higher levels of the general system design. Dynamic linking, demand segmentation, transient processes, and a user domain are not necessary for its intended operation, and are excluded. The software of SASS is partitioned into two domains. The security kernel, which is the most privileged domain, manages system physical resources in a manner designed to prevent unauthorized information flow, regardless of action taken by other elements in the system. The less privileged domain, the supervisor [2], provides each host with a hierarchical file system in which it may store and retrieve files and share them with other hosts. The hosts send commands and transfer files via bidirectional digital links. SASS was designed for implementation of currently available microprocessor hardware. Multiprogramming is used to improve system efficiency and to create a virtual environment which frees the remainder of the operating system from a dependence on the physical processor configuration. Processor management provides a means of coordinating the interaction of the asynchronous processes which comprise the system. This implementation employs a processor multiplexing technique for a distributed kernel and presents a virtual interrupt mechanism. The modular, hierarchical structure of the software is loop-free to support system expension to higher level functions.

Although the primary goal of the design is security, the clean, logical, process-oriented structure of SASS offers other benefits as well, including fault tolerance, resource configuration independence, and efficiency.

B. COMPUTER SECURITY

The need for providing protection for information within a computer system is well documented. Development of the security kernel technology [5,6], has transformed the operating system designer's approach from a game of wits with penetrators into a methodical design process.

In general, security is provided by providing protection for information in accordance with a specific protection policy. In the case of computer security this is accomplished by controlling the access of people to information. Although this protection can be provided by external controls (e.g., confining the computer system and all its users within a physical security perimeter), this method is inefficient and prone to human error. Furthermore, a distributed computer network will probably be dispersed over too wide an area to be physically confined. Supported by the security kernel approach, an internal protection mechanism controlled by the computer operating system is a feasible solution.

1. Reference Monitor

The concept of protection is realized within the computer system by the implementation of a mathematical model of information security. This model is based on an abstract representation of security called the Reference Monitor [7]. The Reference Monitor describes a mechanism for controlling the access of subjects to objects, based on a set of access authorizations (figure 2).

Reference Monitor

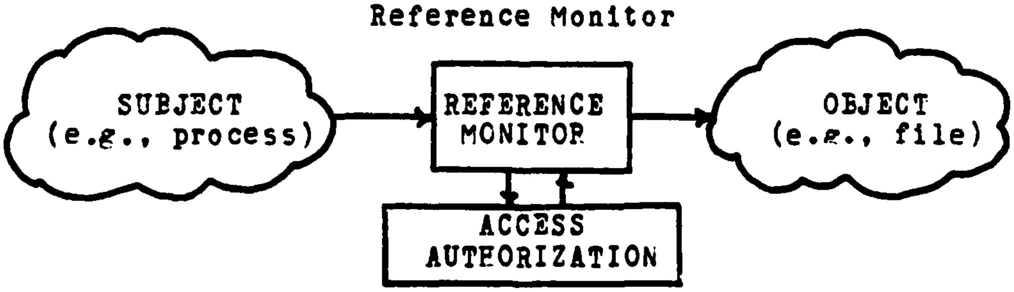


Figure 2

Every time a subject attempts to access an object, the Reference Monitor checks to determine if the subject has authorization to perform the desired operation (e.g., write, read) on the object. If the policy does not authorize the access, the Reference Monitor will prevent the subject from performing the requested operation. This mechanism is realized within the operating system as the security kernel. Several system features are required in order for the mechanism to function correctly.

First, every reference to information (i.e., every access to primary memory by the processor) must go through the security kernel.

Second, the implementation of the security kernel must be an exact representation of the mathematical model of information security.

Third, the security kernel must be tamper-proof.

2. Security Policy

The security policy to be enforced by the computer system consists of external laws, rules, regulations, etc., which establish permissable information access independent of the computer system. Therefore, a computer system will be secure only with respect to a specific security policy. The security kernel concept supports a broad range of security policies that can be divided into two classes, non-discretionary and discretionary security.

a. Non-discretionary Policy

Non-discretionary security policy uses labels to insure only permissable access of subjects to objects is provided. Object labels reflect object sensitivity and subject labels reflect subject authorization. (For example, National Security Policy labels include Unclassified, Secret, etc.). A non-discretionary security policy provides compromise protection (from unauthorized reading), integrity protection (from unauthorized modification), and must prevent information leaks resulting from indirect access to unauthorized information as well. A non-discretionary security policy requires that all subjects and objects have labels. Most contemporary computer systems do not provide this explicit labeling and therefore implicitly make all access permissable.

b. Discretionary Policy

Discretionary security policy provides a finer division of access by allowing individual subjects to decide which of the permissable accesses, determined by non-discretionary policy, will actually be allowed (e.g., DoD's "need to know"). Many contemporary computer systems support discretionary security policy with access control lists, file passwords, capability lists and other mechanisms.

3. Security Kernel Design

By careful interpretation of the mathematical model of the Reference Monitor, the security kernel is designed to be a subset of operating system functions. Kernel primitives form an interface between this subset and the remainder of the system. If these primitives are implemented correctly, their use guarantees that information will be protected in compliance with system security policy, regardless of any action taken by other portions of the operating system or by the user. A more detailed discussion of the security model is provided in [4,5,6].

C. SCOPE OF THESIS

In this chapter a subset of the general operating system design, the Secure Archival Storage System (SASS), was described. The concept of information security was examined and the security kernel was presented as a technically sound approach to the problem of providing internal computer security.

Chapter Two will discuss the design goals of this operating system. Functional design requirements will be developed and the issues of physical resource management and performance will be traced to specific attributes desired in system hardware. The rationale behind the ultimate selection of Zilog's Z8000 Microprocessor and Z8010 memory management unit (MMU) for use in the SASS testbed implementation of this operating system will be discussed.

Chapter Three will describe the high level design of SASS with an emphasis on the security kernel design. A view of the user (computer host) environment as a collection of cooperating processes will be presented, and the hierarchical structure of the distributed kernel modules will be examined in detail.

Chapter Four will present an implementation of the SASS security kernel modules that provide multiprogramming and processor management. The construction of the virtual machine environment will be described and the advantages of a two-level scheduling mechanism will be explained.

Finally an evaluation of this implementation will be presented with recommendations for improving the design and suggestions for follow on work.

II. OPERATING SYSTEMS DESIGN CONCEPTS

The kernel primitives providing multiprogramming and process management form one of the smallest and most basic subsets in the family of operating systems designed by O'Connell and Richardson [4]. As developed here they were implemented specifically to support SASS. In general the same kernel primitives will support all members of this design family.

Before discussing the high level design of the SASS security kernel and presenting an implementation of these primitives, it is useful to investigate the general design methodology applied to the development of this operating system. In this chapter the design goals of SASS will be analyzed and traced to functional requirements and hardware attributes considered necessary or desirable in support of the system's design goals. It is recognized that the operating system user Will probably not address these issues directly when specifying system design goals. The material presented here concerns the approach of the system designer to the definition of requirements implicitly related to user design goals.

A. DESIGN PHILOSOPHY

Two issues confront the operating system designer. First, he must provide system functions which support the services requested by the user. These functional requirements affect the logical design of the system. Second, he must address issues of cost and performance. Cost and other management considerations will not be addressed here. Performance issues concern the management of physical resources and ultimately can be reduced to hardware requirements.

There is a considerable amount of literature devoted to the development of the functional design of operating systems. Dijkstra [8] has described a technique for reducing the complexity of the design by allocating operating system activities to a number of cooperating processes. Process structure is simplified in turn by defining its functions in levels of increasing abstraction and by applying the principles of structured programming.

Madnick and Donovan [9] have described an operating system as a hierarchical extended machine. Program modules are added to the system hardware to provide many extended instructions in addition to the hardware instructions available on the bare machine. In complex systems one extended machine may be constructed upon another to form a system composed of levels of abstract (virtual) machines.

Saltzer [10] and Reed [11, 12] have discussed the advantages of resource virtualization and have described some useful interprocess communication mechanisms. The general design strategies presented in this and other research aid the operating system designer in developing system functions in a clean, logical, verifiable design.

The selection of an appropriate computer architecture, which supports both functional requirements and the efficient management of physical resources, often proves to be a more difficult issue. Frequently operating systems design is shaped by the capabilities of system hardware. This may be a result of performance limitations or cost of available hardware, but often this course is taken because traditionally, system design begins with hardware. Since a primary goal in operating systms design is to create a specific operational environment for the user, it would appear to be preferable to design from the desired environment "down to" the hardware. In this way all components of the system, software and hardware alike, are evaluated in the light of the ultimate goals of the system, and any incompatabilities between required functions and hardware capabilities will be discovered early in the design. Then, if modifications are required, design changes can be made at a high level which will preserve design integrity. LSI technology currently provides a wide variety of relatively inexpensive microprocessor hardware from which to select specific physical components. Furthermore, it is often feasible to design special purpose hardware to specification. So the traditional restrictions on hardware versatility in systems design need not apply in many cases to microprocessor systems.

In summary, the top-down design philosophy can be applied to operating systems design in the following manner:

1. Identify general and specific design goals.
2. Derive functional design requirements.
3. Identify performance requirements.
4. Select system hardware.
5. Develope kernel software.
6. Develope the remainder of the O/S software.

B. GENERAL DESIGN GOALS

Although many design goals depend upon specific system application, there appear to be some attributes desirable in all operating systems.

1. Logical Structure

Computer system design is an engineering problem and the tools of the engineering design process should be applied to the development of software as well as hardware [13]. Clarity should be a major goal of any design for if the operating system cannot be understood easily it will be difficult to test, difficult to maintain, and its correctness will always be in doubt. A sound engineering design philosophy is not guaranteed to generate error free systems, but if system functions are cleanly organized and well understood, then it is likely that there will be few errors and these can be corrected without difficulty when discovered.

2. <u>Fault Tolerence</u>

If an operating system is to be reliable, the software it uses must be protected from damage whenever possible. In particular, tasks performed by the system should be isolated from another so that a malfunction (e.g., as the result of hardware failure) in one task has no effect on others.

3. <u>Efficiency</u>

The efficient use of physical resources (processors, memory, periphals, etc.) continues to be a primary design goal. However, since hardware is no longer the scarce, expensive commodity it once was, a concern for overall system efficiency (i.e., higher thorugh-put, faster response time) may be more important. With appropriate component selection many software functions can be replaced by hardware functions that can provide an improvement in system performance at a small additional hardware expense.

C. SPECIFIC DESIGN GOALS

The family of operating systems designed by O'Connell and Richardson provides all of the services expected of a state of the art, general purpose operating system. Many of these general services are not necessary in the SASS subset of the family. The number of processes required by SASS is determined by the number of host computers linked to SASS hardware. A design choice was made to fix this number at system generation time. Therefore dynamic process management is not required; SASS processes exist for the life of the system. A primary function of SASS is the transfer of files between host computers and SASS via bidirectional digital links. As a result, the system will have a low transaction rate, and the relatively fast response time desired in a time-sharing system is not required here. Sass does not provide programming services to users; the system strictly manages an archival storage system. This eliminates the requirement for a user domain and because the demands on primary memory are not excessive, there is no need for dynamic memory management.

Other services of the general system provide essential support to SASS. These services include I/O management, file management, and the physical resource management and information protection functions provided by the security kernel.

The SASS requirement to provide multiple host computers (users) with controlled, shared access to a multilevel secure "data warehouse" leads to several design goals. These include: internal security to proctect information in a distributed computer network; configuration independence for system versatility; and a subsetting capability to support future system expansion to more complex members of the design family.

1. Internal Security

A unique feature of SASS is the specification of multilevel security as a primary design goal. Multilevel security provides controlled sharing of information of varying sensitivity among many users in accordance with an access policy implemented internally by the operating system. It is essential that a system supporting a remotely accessed data base containing information of different access classes be provided with an internally enforced security policy.

2. Configuration Independence

The resource configuration of a multicomputer system is highly changeable. Processors are added and removed; memory is reconfigured; interconnection schemes are altered and peripherial equipment is changed. The operating system of such a design should be sufficiently flexible to permit maintenance and to allow for growth and reconfiguration without requiring drastic system redesign or noticeably affecting the user's environment.

3. Sub-setting Capability

Operating system "sub-setting" refers to the ability to form meaningful subsets of the design by eliminating many of the services that can be provided by the system without affecting the usefulness of the remainder of the system. Sub-setting permits the system to be tailored to fit a number of specific designs ranging from a simple monitor to a full service time-shared computer utility. The implementation presented in this thesis creates a monitor that provides multiprogramming and processor management. This subset supports more complex family members of the design such as SASS.

D. DESIGN REQUIREMENTS

In a top-down approach to design, goals are clarified and defined by requirements which describe either the system functions or address cost and performance issues (hardware requirements). The functional requirements defined below support the specific design goals of SASS and provide features desirable in any operating system, such as a logical structure, fault tolerance, and efficiency of operation.

1. Functional Requirements

Functional requirements define services which must be provided to support the user's environment.

a. Process Organization

By designing an operating system as a collection of cooperating processes, system complexity can be greatly reduced [8]. This is because the asynchronous nature of the system can be structured logically by representing each independent, sequential task as a process and by providing interprocess communication mechanisms to prevent races and deadlocks during process interactions.

The notion of a process provides a complete description of all instructions executed and all memory locations referenced during the performance of a task. A process is defined by an address space and an execution point. The address space is the set of memory locations which could be accessed during process execution. (The process is viewed as a past, present and future "history" of memory locations which actually were referenced.) The execution point is the state of the processor at a given instant during process execution. In the abstract view, an address space is defined by a collection to discrete points, each representing a memory word. The process is described by the path traced through this address space from process creation to destruction. In figure 3 the main path traces the process execution point as it moves from one instruction (i.e., memory word) to another during process execution. The branches from this execution point path represent data references.

Process History

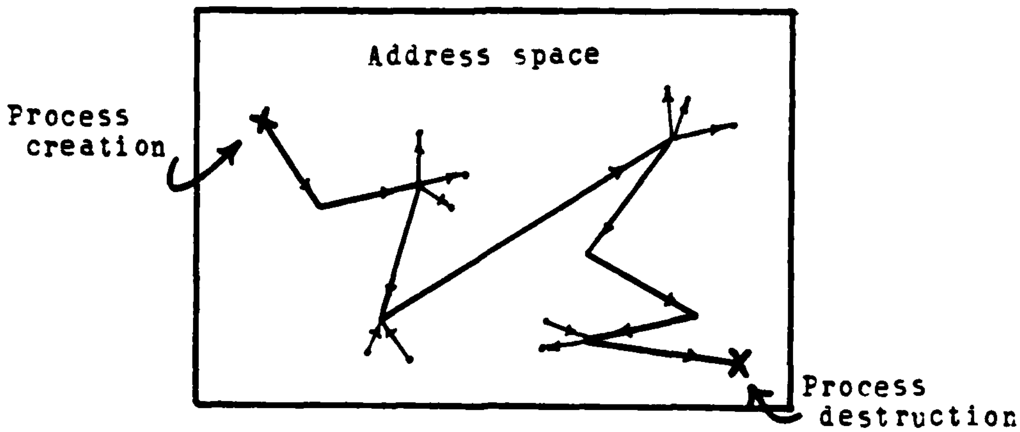


Figure 3

Several advantages result from using a process oriented design. As a tool for dealing with the asynchronous nature of system operation, processes provide a simple, logical, high-level structure for the design. For example, the Secure Archival Storage System supports each host with three processes: a I/O Manager, a File Manager, and a Memory Manager, which interact to provide secure file management services to the host. This interaction will be described further in the next chapter. Since each process is confined to a secific address space, tasks are isolated from one another and system fault tolerance is improved. By providing an internal representation for each user, a process nicely fits the definition of a "subject" in the Reference Monitor and therefore supports the design goal of providing internal security.

b. Memory Segmentation

The address space of a process is composed of a collection of segments. A segment is a logical collection of information (e.g., procedure, data structure, file, etc.) and is the basic logical object of this design. Figure 4 illustrates the two-dimentional nature of the segment address. Each segment consists of an arbitrary region of memory containing a sequence of words with conventional linear addresses. Two-dimentional addressing frees information from dependence on a particular memory location by making it arbitrarily relocatable.

Segmented Addressing

<<SEG #n>> OFFSET

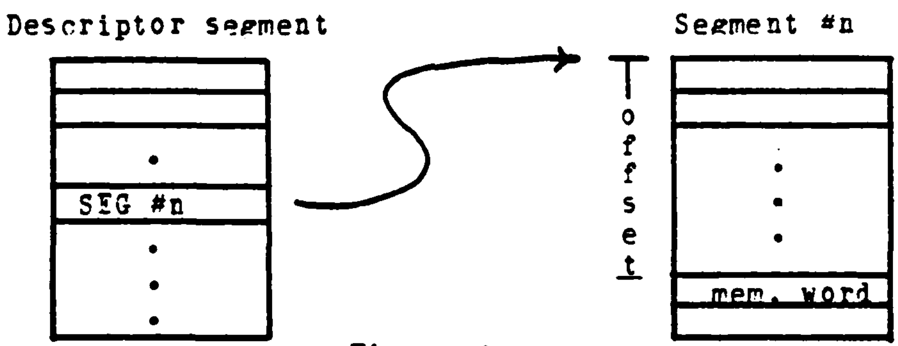


Figure 4

The descriptor segment provides a list of descriptors for all segments in a process address space. In addition, segmentation supports information sharing since a segment may belong to more than one address space.

Segmention also provides a means of associating logical attributes and labels with each segment, such as access class, domain, etc. This feature supports segments as internal representations of the Reference Monitor's "object".

c. Abstraction

Abstraction provides a method for reducing problem complexity by applying a general solution to a collection of specific cases [14]. Structured programming provides a tool for creating abstraction in software design. By strictly applying two special rules in addition to the general principles of structured programming, a structure consisting of levels of increasing abstraction can be constructured.

First, calls cannot be outward toward higher levels of abstraction. This frees lower levels from a dependence on higher levels by creating a loop-free structure [15] and results in a design which is capable of having subsets.

Second, calls to lower levels must be by special entry points or gates. Each level of abstraction creates an virtual hierarchical machine [9]. The gate to each level provides a set of instructions created for that virtual machine. Thus higher levels may use the resources of lower levels only by applying the instruction set of a lower level machine. (At domain boundaries, use of gates is strictly enforced by a ring-crossing mechanism; otherwise gate use is implicit in the structure of the software.) Once a level of abstraction has been created, the details of its implementation are no longer an issue. Instead users see layers of virtual machines , each defined by its extended instruction set.

Each process used in SASS is designed in levels of abstraction. When the rules of abstraction are applied to level Ø, the physical resources of the system, these resources are "virtualized". Thus the first level of abstraction creates "virtual processors", "virtual memory", and "virtual devices" from the system's hardware. At each higher level the detail of the design is reduced. The gate at the boundary between the highest level of the security kernel and the lowest level of the supervisor provides a mechanism for isolating the kernel as well as insuring that each memory access is via kernel software. This mechanism is implemented in SASS by a ring-crossing mechanism called the Gatekeeper.

d. Resource Virtualization

The first levels of abstraction above system hardware create virtual representations of physical resources (virtual processors, virtual memory, virtual periphals). Since upper levels of the design operate on these virtual resources, rather than on physical resources, most of the design (i.e., everything above resource virtualization levels) is independent of the physical configuration of the system. By providing virtual to real resource binding in the kernel, and by enforcing entry into kernel levels with the Gatekeeper, SASS protects physical resources from tampering and insures memory access only via the kernel. As a result, the kernel modules of each process will guarantee that the system's non-discretionary security policy is enforced. Including in the kernel only those functions essential to system security keeps it small and reduces the job of verification to manageable proportions.

2. Hardware Requirements

Virtual resources are created by the multiplexing of various types of information on a physical resource. Multiplexing can be defined as the use of a single resource for different purposes at different times. For example the physical bus lines can be used both for addresses and data during different times during the machine cycle. Similarly, logical users of a hardware system can share resources. The ability to multiplex processors and memory efficiently provides a mechanism for the virtualization of these physical resources.

a. Processor Virtualization.

A virtual processor is a data structure that contains a complete description of a process in execution on a physical processor at a given instant. This description is contained in the process execution point. The address space of the process must be accessable to the virtual processor when it is loaded on (bound to) a CPU. To provide a useful virtualization capability, the CPU must have the ability to efficiently multiplex process exection points and address spaces (i.e., it must support multiprogramming).

b. Memory Virtualization.

In many memory handling schemes Process cannot run unless the entire address space is loaded in primary memory. This may require a large main memory or it may restrict the size of the address space. An alternative plan requires an 'operating system which manages primary and secondary memory to create the illusion of a memory which is larger than the system's primary memory. Since the larger memory is only an illusion, it is often called virtual storage. The logical, relocatable, information objects created by memory segmentaion, provide an essential memory multiplexing mechanism for the efficient implementation of virtual storage.

c. Protection Domains

An essential requirement of internal security is that the security kernel be isolated from other elements of the system. This can be accomplished by the construction of protection domains. Protection domains are used to arrange process address spaces into rings of different privilege. This arrangement is a hierarchical structure in which the most priviledged domain is the innermost ring. The structure essentially divides the address space into levels of abstraction with strictly enforced gates at the ring boundaries (Figure 5).

SASS Protection Rings

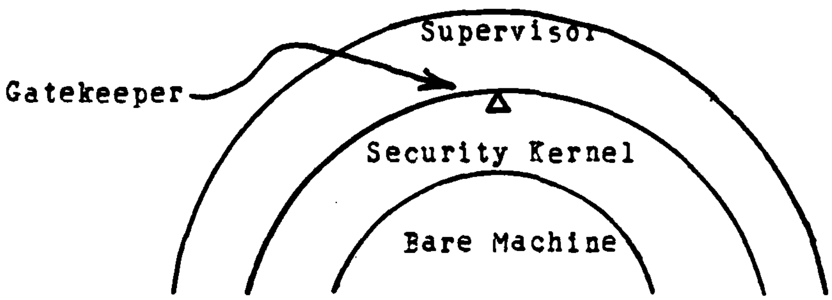


Figure 5

Protection rings may be created in software, but a hardware implementation, where gate use is enforced by hardware, is much more efficient [16].

The protection provided by the ring structure is not a security policy. (Security protection is implemented by a lattice structure known to the Non-discretionary Security module in the kernel.) It does, however, enforce the hierarchy of the virtual machine by creating a privileged kernel ring within the supervisor ring.

E. HARDWARE SELECTION

The manifestation of an operating system design is, of course, software in execution on system equipment. If system equipment must be selected early in the design, care must be taken to insure that overall system design goals are compatible with actual hardware capabilities. If design goals must be met (e.g., the enforcement of internal security in SASS), then actual hardware selection should be made late in the design process. Then, even if a poor hardware choice is made, the penalty for correcting it will be small, since only the lowest level of the design (where resources are virtualized) need be changed. In any case the design of the operating system and the design or selection of system hardware must proceed in concert.

1. <u>Zilog Z8001</u>

The Z8001 is a general purpose 16-bit microprocessor [17] with an architecture which supports memory segmentation and two-domain operations. It was selected as the target machine for implementation of the system because of the full range of support and close match it provided to design requirements. These supporting features are described below.

a. Memory Segmentation

The CPU can directly access 8M bytes of address space using a memory segmentation capability provided externally by a Memory Management Unit (Z8010 MMU). The 23-bit address required to address 8M bytes is a logical two dimensional address consisting of a 7-bit segment number and a 16-bit offset. The memory management unit converts this into a 24-bit address for the physical memory. The address space can be divided into as many as 128 relocatable segments containing up to 64K bytes each. Each memory segment can be assigned several attributes which provide memory access protection (read only , system mode only (i.e., ring #), execute only, etc.) and memory management data (changed, referenced). With these capabilities the Z8001 CPU can support all requirements for segmentation, memory virtualization and protection domains.

b. Multiprogramming

Processor multiplexing is supported by the CPU's multiprogramming capabilities. MULTI-MICRO instructions aid in establishing a synchronization mechanism (by mutual exclusion) between multiple processors. Seperate stack, data and code address spaces are maintained for each ring of operation. The load multiple instruction allows the contents of registers to be saved and loaded efficiently. These features permit efficient storing and loading of process execution points.

Address space multiplexing is also supported but is somewhat inefficient. In some systems, such as Multics [18], a descriptor base register (DBR) is provided to point to a process descriptor segment in memory, so changing the address space of the physical processor is accomplished merely by changing the DBR. Since the Z8001 CPU implements the descriptor segment as a collection of descriptor registers in the MMU, all of the descriptors for the address space must be saved and loaded to change processes. This can make processor multiplexing (multiprogramming) quite inefficient. In the worst case, when the entire MMU is saved and loaded, a process switch will take about 2 ms. It may be possible to improve on this performance by increasing the number of MMU's in the system. Then the address space can be changed simply by switching control to another MMU.

c. Two-Domain Operations

The Z8001 CPU can operate in either system mode or normal mode. In the system mode all operations are allowed, but in the user mode, certain system instructions are prohibited. The system call instruction allows controlled entry to the system mode. This two-domain instruction capability supports the two domain sturcture of SASS by providing a single controlled entry into the kernel (SYSTEM CALL instruction). The descriptors contained in the MMU registers provide the capability to partition process address spaces into supervisor and kernel domains.

2. Selection Rationale

The characteristics listed above - processor multiplexing support, a memory segmentation capability, multiple domain insturctions, and multiple domain memory partitioning - are features which are essential to an efficient implementation of SASS. The Z8001 has other desirable features: vectored and non-vectored interrupts, large, powerful instruction set, many data types, etc. These attributes make the Zilog system a suitable choice as a bare machine for the Secure Archival Storage System.

F. SUMMARY

This chapter has provided a description of the methodology employed in the design and specification of SASS. In particular it was noted that a top-down design philosophy most effectively supported implementation of system design goals. Requirements supporting the primary design goal of internal security and other general and specific goals were defined and traced to desired hardware capabilities. Finally, capabilities of Zilog's Z8001 microprocessor which support the SASS design were described.

Chapter Three will provide an overview of the SASS design. The design will be described from a process viewpoint and the hierarchical structure of the distributed kernel will be examined.

III. SECURITY KERNEL DESIGN

The high level design of the Secure Archival Storage System can be described by a collection of cooperating processes. The use of processes to perform operating system functions greatly simplifies the problem of describing the asynchronous manner in which services are requested.

A. PROCESS VIEW

There are two kinds of processes within SASS, supervisor processes and kernel processes. Supervisor processes provide high level services to host computers [2]. Certain functions of the operating system are distributed throughout all of these processes; that is, supervisor processes logically share a collection of distributed kernel modules. Kernel processes provide specialized services within the operating system. The system user is not aware of the existence of these processes, but they are called upon, within the kernel domain, by supervisor processes to perform necessary operating system functions in support of user services.

1. Supervisor Processes

One pair of supervisor processes, an I/O Manager and a File Manager, represents each computer host supported by SASS.

The File Manager controls SASS and directs all interaction between SASS and computer hosts in order to maintain a structure of hierarchical files on behalf of each host It interprets commands received from hosts via the I/O Manager and coordinates the execution of requested services with assistance from the I/O Manager and the Memory Manager (described below).

The I/O Manager transfers information via a link between each host and SASS. Data is transfered by fixed-size packets in command, data, and synchronization formats. The I/O Manager provides only a transfer service and does not interpret the data.

2. Kernel Processes

The two kernel processes used by SASS are the Memory Manager and the Idle process. The Memory Manager controls primary and secondary memory. The design of this process is the topic of concurrent thesis research [3]. The Memory Manager transfers segments between primary and secondary memory in response to requests from supervisor processes.

The Idle process defines the "no work" state of the system. SASS attempts to schedule useful work on system processors whenever possible. Only when there is no work to be done, (i.e., no commands pending from hosts) will this process be called upon to execute.

3. Host Environment

Host computers view SASS as a remote data warehouse where they may store and retrieve files (figure 6). Each host is provided with a virtual file hierarchy constructed from directory and data files. A pair of SASS supervisor processes (an I/O Manager and a File Manager) provide each host with a set of commands by which it may store and retrieve files in its virtual file system and share files with other hosts. The distributed kernel functions of each process control the physical resources of the system in support host commands and SASS security policy.

SASS Process Configuration

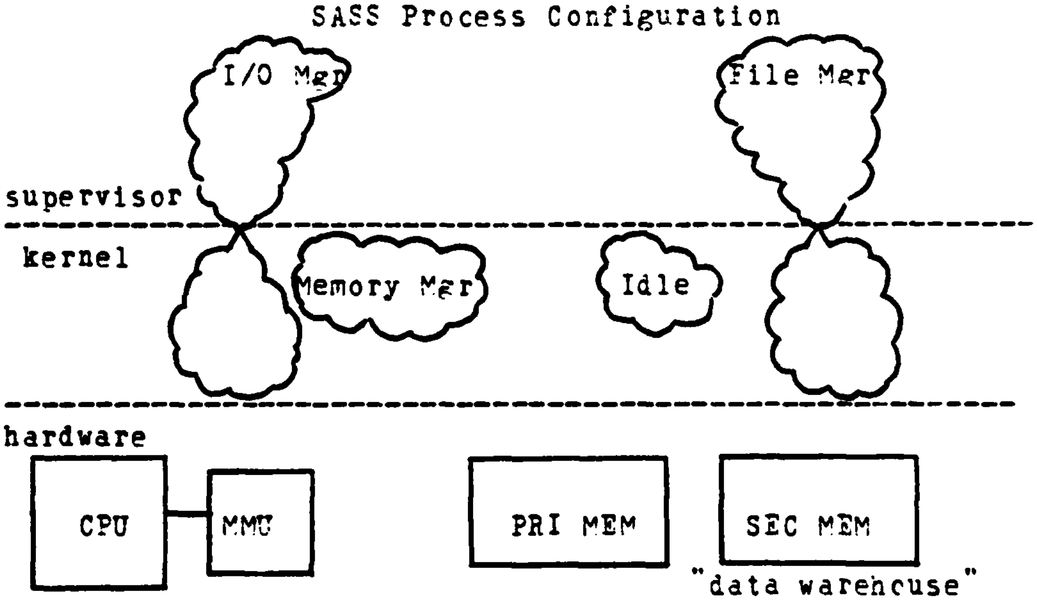


FIGURE 6

B. VIRTUAL MACHINE VIEW

The distributed modules of the security kernel create a virtual hierarchical machine which controls process interactions and manages physical processor resources. The kernel is not aware of the details of process tasks. It knows each process only by a name (viz., an entry number in a table) and provides processes with scheduling and interprocess communication services based on this process identifier. All supervisor processes share the modules of this virtual hierarchical machine (Figure 7).

The kernel is constructed in layers of abstraction. Each layer, or level, builds upon the resources created at lower levels. The rules of abstraction described in Chapter 2 were applied to the design of this structure. Level 0 is the bare machine which provides the physical resources (processors and storage) upon which the virtual machine is constructed. The remainder of this chapter will describe the level of virtualization (or layer of abstraction) created by each distributed kernel module.

1. Inner Traffic Controller Module

Level-1 of this virtual machine is the Inner Traffic Controller Module. This module creates a set of virtual processors with the extended instruction set: SIGNAL, WAIT, SWAP_VDBR, IDLE, SET_VPREEMPT, TEST_VPREEMPT, and RUNNING_VP.

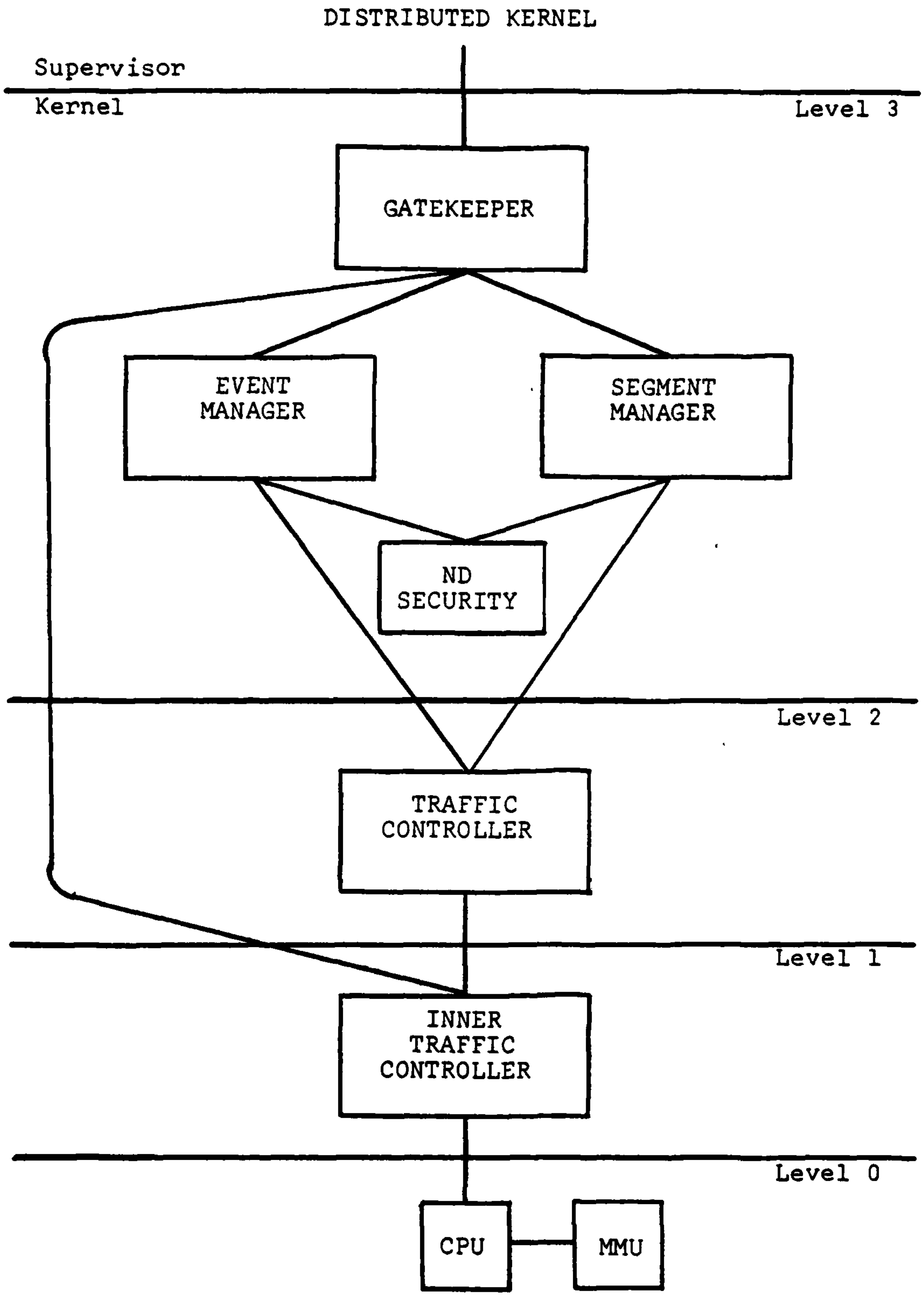

DISTRIBUTED KERNEL
Supervisor
Kernel
Level 3
GATEKEEPER
EVENT MANAGER
SEGMENT MANAGER
ND SECURITY
Level 2
TRAFFIC CONTROLLER
Level 1
INNER TRAFFIC CONTROLLER
Level 0
CPU
MMU

Figure 7

SIGNAL and WAIT provide an interprocessor communication mechanism used within the kernel to provide multiprogramming. These instructions invoke the level-1 scheduling procedure, GETWORK, which multiplexes virtual processors on a physical processor.

SWAP_VDBR and IDLE are instructions invoked from level-2 by the Traffic Controller Module to schedule processes on a virtual processor.

SET_VPREEMPT and TEST_VPREEMPT create a virtual processor interrupt mechanism. SET_VPREEMPT is invoked from level-2 when the traffic controller desires to load a new process on a virtual processor that is not scheduled. TEST_VPREEMPT is invoked by the Gatekeeper of each distributed process upon every exit from the kernel domain. The Gatekeeper unmasks virtual interrupts by testing the interrupt flag of the scheduled virtual processor. If the flag is set, a virtual interrupt handler is invoked, otherwise the process enters the supervisor domain normally.

RUNNING_VP is invoked from level-2 to provide the Traffic Controller with the identity of the currently scheduled virtual processor. The identity of a particular processor must be known in the virtual environment, just as the identity of a physical processor is required in a multiprocessor system.

2. Traffic Controller Module

The Traffic Controller resides at level-2. It manages the scheduling of processes on virtual processors by invoking the extended instructions of the virtual processors in level-1. In addition to implementing the level-2 scheduling algorithm, the Traffic Controller creates the extended instruction set: ADVANCE, AWAIT, and PROCESS_CLASS.

ADVANCE and AWAIT are used to implement eventcounts and sequencers [11], an inter-processor communication (IPC) mechanism invoked by the supervisor. Although SIGNAL and WAIT provided an adequate interprocessor synchronization mechanism within kernel, Parks [2] determined that supervisor process synchronization would be more effectively served in the secure environment of SASS by the use of eventcounts.

PROCESS_CLASS is invoked from level-3. It returns the label, subject access class, of the current process for determining a subject-object relation.

a. Scheduling

Scheduling functions are divided between the Inner Traffic Controller and the Traffic Controller. The Inner Traffic Controller multiplexes virtual processors on a CPU. The Traffic Controller schedules processes on virtual processors.

The division of the scheduling algorithm between these two levels simplifies its design, because it seperates the issues of virtual processor management (multiprogramming) from virtual memory management [12]. A design choice was made to provide each system CPU with a small fixed set of virtual processors. Since the virtual processor data base is shared by all system CPU's, it must remain permaently in global memory.

The process data base, used to implement level-2 scheduling will be much larger. Since supervisor processors are known to the entire system, this data must also be kept in global memory. Because level-2 is subject to memory management, this data could be kept on secondary storage and moved to primary memory when requested.

SASS does not provide dynamic memory management, therefore the two-level scheduling design presented here is not essential to the design. However, the structure has been provided in this implementation to support more complex family members of the O'Connell-Richardson design. Figure 8 illustrates the two levels of scheduling employed by the distributed kernel.

The two virtual processors (Mem_Mgr_VP and Idle_VP in Figure 8) are permanently bound to kernel processes and are not in contention for process scheduling. The remaining VP's are temporarily bound to supervisor processes as determined by the Traffic Controller. If no supervisor process is available, the Traffic Controller invokes the Inner Traffic Controller (IDLE) which loads an Idle process on the virtual processor.

The Inner Traffic Controller schedules virtual processors on the physical processor. Ready virtual processors with temporarily bound idle processes (VP #1 and VP #2 in Figure 8) will be scheduled only to give an Idle process away for a supervisor process (i.e., when virtual preempt flag is set). The Idle process will actually run when the virtual processor to which it is permanently bound (the Idle-VP in Figure 8) is scheduled. This will happen only when all other VP's are waiting or temporarily bound to Idle processes, i.e., when there is no useful work for the CPU.

TWO-LEVEL SCHEDULING

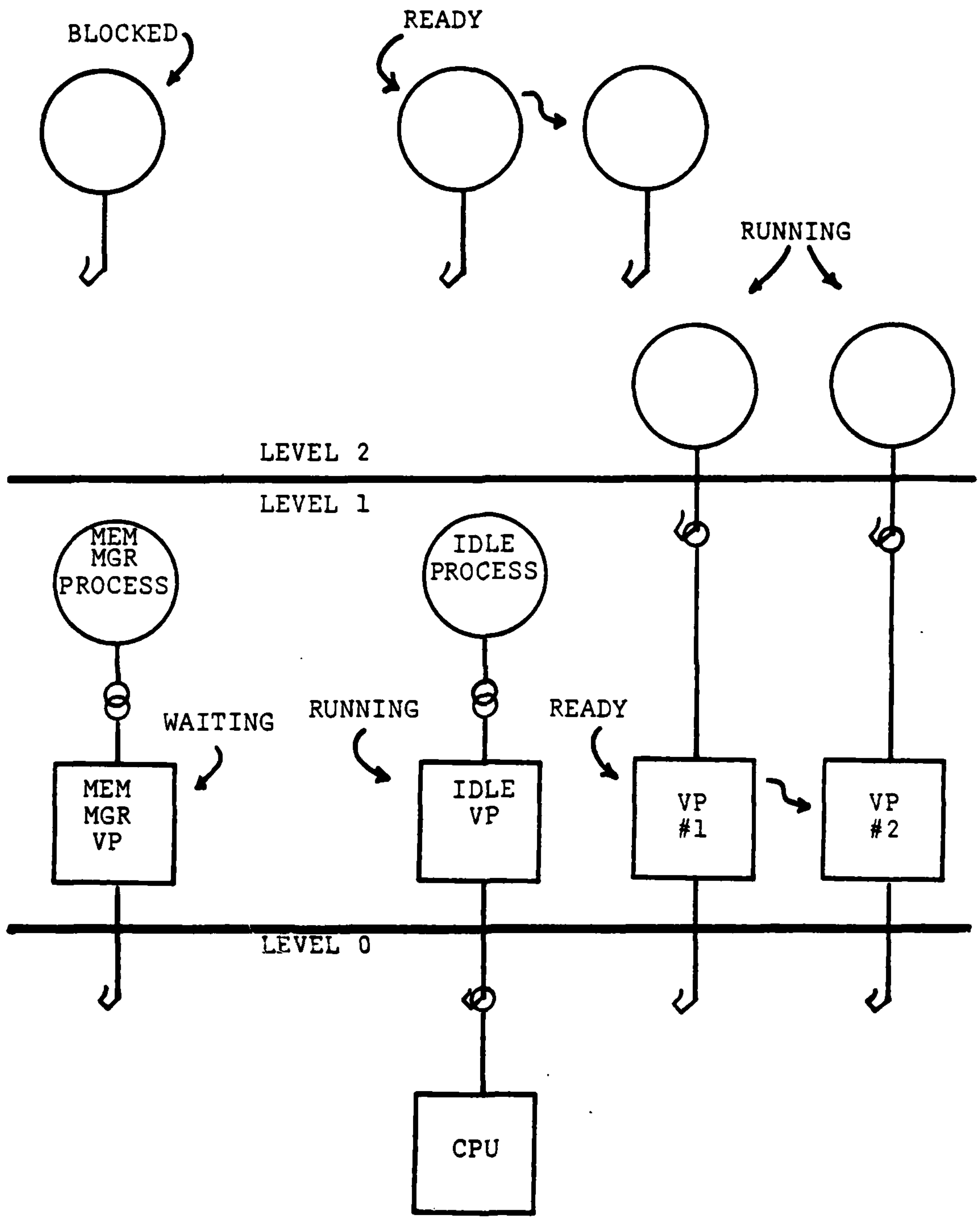


Figure 8

3. Non-Discretionary Security Module

The Non-Discretionary Security module in level-3 reflects the system's security policy. It compares two labels, subject and object access classses, passed to it by other modules, and returns the relationship of the labels based on a lattice structure known to it. To perform this function it provides the extended instruction, RELATION, which is used by the Event Manager and the Segment Manager to determine access permission. These modules make decisions about access based on the relationships: equal, less than, greater than, and not related. The Non-discretionary Security module is the only module which interprets the labels themselves. A different security policy (e.g., Privacy Act vs DOD) can be implemented simply by changing the lattice structure used in this module.

4. Event Manager Module

The Event Manager is a level-3 module invoked by supervisor processes via the gatekeeper. This module creates a set of extended instructions: ADVANCE, AWAIT, READ and TICKET. It determines the access permission of desired interprocess communications and obtains a global handle from a Memory Manager data base where event data is stored. If access is permitted, the event manager passes this handle, which identifies the event, to the Traffic Controller where the appropriate event count instruction is invoked. For sequencer operations the Memory Manager is invoked directly.

The use of the handle is necessary because of the design choice to store event data in a data base of the Memory Manager [3]. This insures that inter-domain IPC does not violate SASS security policy.

5. Segment Manager Module

The Segment Manager also resides in level-3. This module creates a set of extended instructions for manipulating segments. These instructions are: CREATE, DELETE, SWAP_IN, SWAP_OUT, MAKE_KNOWN, and TERMINATE. Modules of the supervisor domain invoke these instructions to coordinate host support. CREATE and DELETE add and remove segments from the system. SWAP_IN and SWAP_OUT cause a segment to be moved between primary and secondary memory (i.e., between a paged disk and contiguous memory). MAKE_KNOWN and TERMINATE add and remove a segment from a process address space.

6. Gatekeeper Module

The Gatekeeper exists on the boundary between the kernel and supervisor domains. It provides the sole entry point into the kernel domain, so when the execution point of a process enters the kernel domain of its address space it must do so through the Gatekeeper.

The hardware of the MMU partitions process address spaces into two domains by setting the ring number (zero or one) in each segment's attribute register. Software provided by the Gatekeeper performs the following additional functions:

Kernel Entry

1. Unmask Hardware interrupts.

2. Save supervisor domain registers.

3. Save supervisor stack pointer in kernel stack segment.

4. Check arguments and invoke appropriate kernel entry points.
(Virtual machine instructions).

Kernel Exit

1. Invoke TEST_VPREEMPT
(i.e., unmask virtual interrupts).

2. Restore supervisor domain stack pointer.

3. Restore supervisor domain registers.

4. Unmask hardware interrupts.

5. Return to process execution point in
in supervisor domain.

C. REVIEW

This chapter has described the high level design of the Secure Archval Storage System kernel from two points of view. In the process view the system is composed of pairs of supervisor processes (an I/O Manager and a File Manager) for each host computer and a pair of kernel processes (a Memory Manager and an Idle process) for each real processor in the system. The supervisor processes provide high level services to host computers while the kernel processes control system memory resources and provide an idle system state. Distributed kernel functions implement two levels of scheduling, provide interprocessor synchronization and communication, manage segments, and isolate and protect the kernel domain of process address spaces. The distributed kernel is constructed as a hierarchical virtual machine. Evidence of the versitility of the loop-free, configuration independent structure of this design can be observed in concurrent thesis work in this area [19]. An Intel 8086 multiprocessor operating system implementation, based on the same design, uses essentially the same virtual insturction set described in this chapter. An implementation of the first two levels of this kernel machine is presented in the next chapter.

IV. IMPLEMENTATION

Implementation of the distributed kernel was simplified by the hierarchical structure of the design for it permitted methodical bottom-up construction of a series of extended machines. This approach was particularly useful in this implementation since the bare machine, the Z8000 Developmental Module, was provided with only a small amount of software support.

A. DEVELOPMENTAL SUPPORT

A Zilog MCZ Developmental System provided support in developing Z8000 machine code. It provided floppy disk file management, a text editor, a linker and a loader that created an image of each Z8000 load module.

A Z8000 Developmental Module (DM) provided the necessary hardware support for operation of a Z8002 non-segmented microprocessor and 16K words (32K bytes) of dynamic RAM. It included a clock, a USART, serial and parallel I/O support, and a 2K PROM monitor.

The monitor provided access to processor registers and memory, single step and break point functions, basic I/O functions, and a download/upload capability with the MCZ system.

Since a segmented version of the processor was not available for system development, segmentation hardware was simulated in software as an MMU image (see Figure 9). Although this data structure did not provide the hardware support (traps) required to protect segments of the kernel domain, it preserved the general structure of the design.

MMU_IMAGE

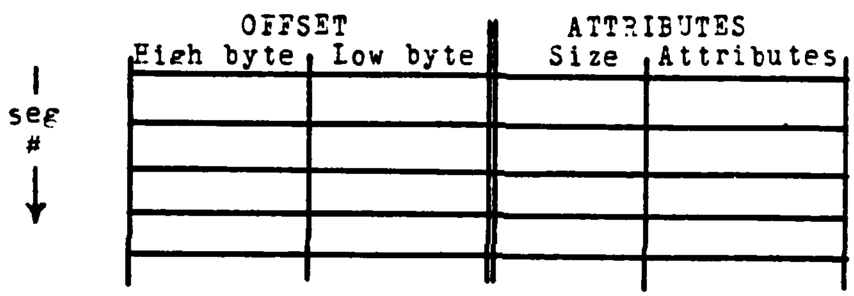


Figure 9

B. INNER TRAFFIC CONTROLLER

The Inner Traffic Controller runs on the bare machine to create a virtual environment for the remainder of the system. Only this module is dependent on the physical processor configuration of the system. All higher levels see only a set of running virtual processors. A kernel data base, the Virtual Processor Table is used by the Inner

Traffic Controller to create the virtual environment of this first level extended machine. A source listing of the Inner Traffic Controller module is contained in Appendix A.

1. Virtual Processor Table (VPT)

The VPT is a data structure of arrays and records that maintains the data used by the Inner Traffic Controller to multiplex virtual processors on a real processor and to create the extended instruction set that controls virtual processor operation (see Figure 10). There is one table for each physical processor in the system. Since this implementation was for a uniprocessor system (the Z8000 DM), only one table was necessary.

Virtual Processor Table

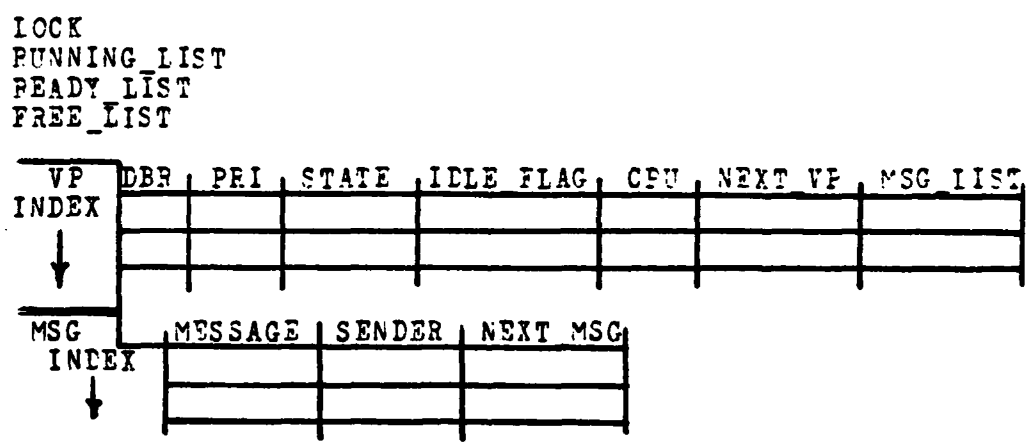


Figure 10

The table contains a LOCK which supports an exclusion mechanism for a multiprocessor system. It was provided in this implementation only to preserve the generality of the design.

The Descriptor Base Register (DBR) binds a process to a virtual processor. The DBR points to an MMU_IMAGE containing the list of descriptors for segments in the process address space.

A virtual processor (VP) can be in one of three states: running, ready, and waiting (figure 11).

Virtual Processor States

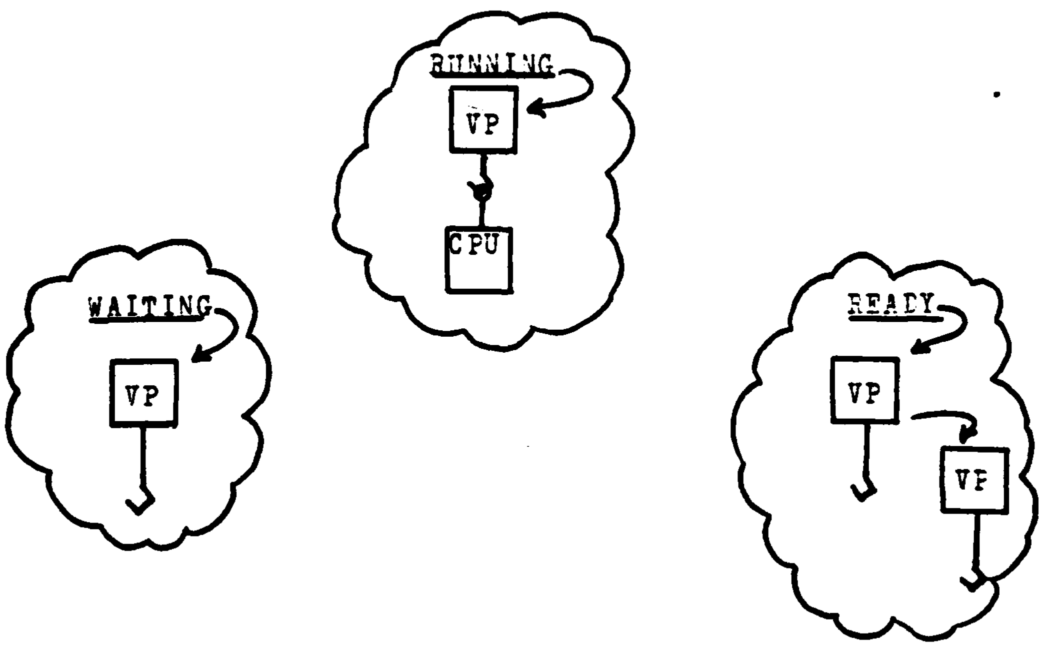


FIGURE 11

A running VP is currently scheduled on a real processor. A ready VP is ready to be scheduled when selected by the level-1 scheduling algorithm. A waiting VP is awaiting a message from some other VP to place it in the ready list. In the meantime it is not in contention for the real processor.

2. Level-1 Scheduling

Virtual processor state changes are initiated by the inter-virtual-processor communication mechanisms, SIGNAL and WAIT. These level-1 instructions implement the scheduling policy by determining what virtual processor to bind to the real processor. The actual binding and unbinding is performed by a Processor switching mechanism called SWAP_DBR [10]. Processor switching implies that somehow the execution point and address space of a new process are acquired by the processor. Care must be taken to insure that the old process is saved and the new process loaded in an orderly manner. A solution to this problem, suggested by Saltzer [10], is to design the switching mechanism so that it is a common procedure having the same segment number in every address space.

In this implementation a processor register (R14) was reserved within the switching mechanism for use as a DBR. Processor switching was performed by saving the old execution point (i.e., processor registers and flag control word), loading the new DBR and then loading the new execution point. The processor switch occurs at the instant the DBR is changed (see figure 12). Because the switching procedure is distributed in the same numbered segment in all address spaces, the "next" instruction at the instant of the switch will have the same offset no matter what address space the processor is in. This is the key to the proper operation of SWAP_DBR.

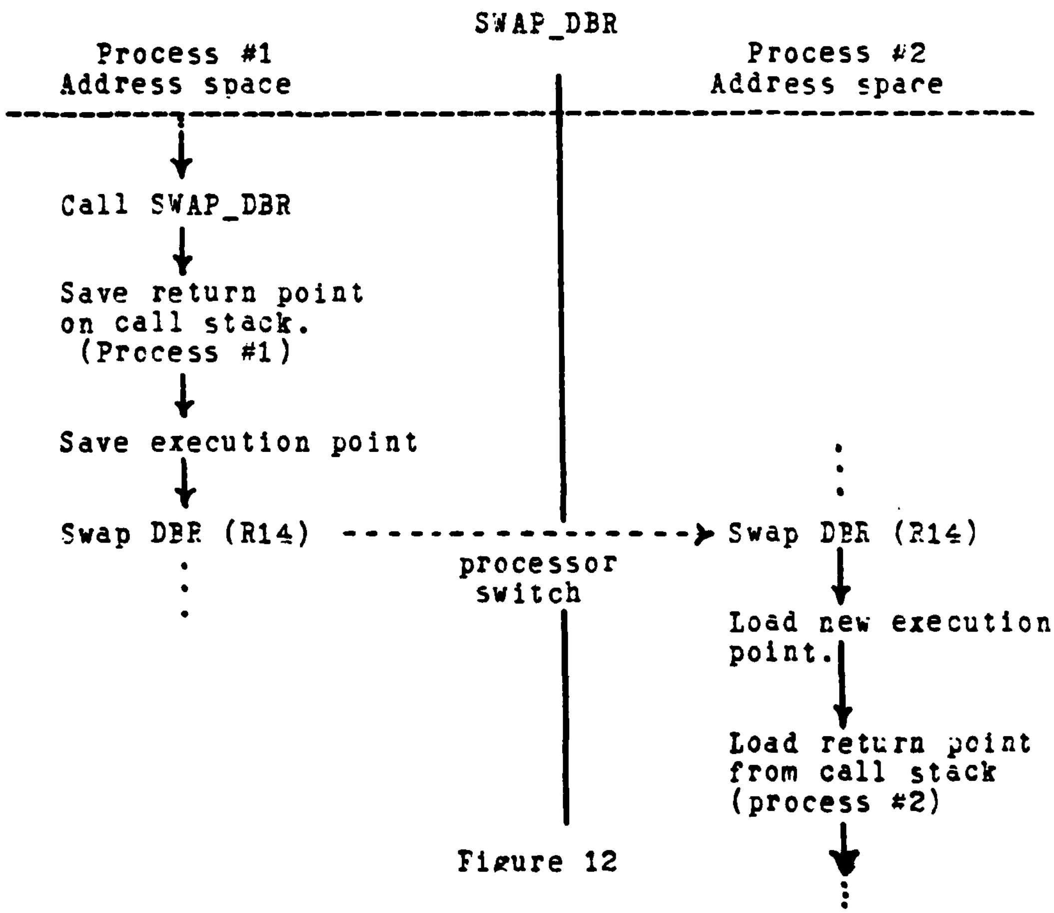


Figure 12

To convert this switching mechanism to segmented hardware it is necessary merely to replace SWAP_DBR with special I/O block-move instructions that save the contents of the MMU in the appropriate MMU_IMAGE and load the contents of the new MMU_IMAGE into the MMU.

a. Getwork

SWAP_DBR is contained within an internal Inner Traffic Controller procedure called GETWORK. In addition to multiplexing virtual processors on the CPU, GETWORK interprets the virtual processor status flags, IDLE and PREEMPT, and modifies VP scheduling accordingly in an attempt to keep the CPU busy doing useful work.

There are actually two classes of idle processes within the system. One class belongs to the Traffic Controller. Conceptually there is a ready level-2 idle process for each virtual processor available to the Traffic Controller for scheduling. When a running process blocks itself, the Traffic Controller schedules the first ready process. This will be an idle process if no supervisor processes are in the ready list.

The second class of idle process exists in the kernel. The kernel Idle process is permanently bound to the lowest priority virtual processor.

The distinction is made between these classes because of the need to keep the CPU busy doing useful work whenever possible. There is no need for GETWORK to schedule a level-2 idle process that has been loaded on a virtual processor, because the idle process does no useful work. The virtual processor IDLE_FLAG indicates that a virtual processor has been loaded with a level-2 idle process. GETWORK will schedule this virtual processor only if the PREEMPT flag is also set. The PREEMPT flag is a signal from the Traffic Controller that a supervisor process is now ready to run.

When GETWORK can find no other ready virtual processors with IDLE and PREEMPT flags off, it will select the virtual processor permanently bound to the kernel Idle process. Only then will the Idle process actually run on the CPU.

Getwork contains two entry points. The first, a normal entry, resets the preempt interrupt return flag. (R2 is reserved for this purpose within GETWORK.) The second, a hardware interrupt entry point, contains an interrupt handler which sets the preempt interrupt return flag. The DBR (R14) must also be set to the current value by any procedure that calls GETWORK in order to permit the SWAP_DBR portion of GETWORK to have access to the scheduled process's address space. Upon completion of the processor switch, GETWORK examines the interrupt return flag to determine whether a normal return or an interrupt return is required.

The hardware interrupt entry point in GETWORK supports the technique used to initialize the system. Each process address space contains a kernel domain stack segment used by SWAP-DBR in GETWORK to save and restore VP states. For the same reason that SWAP-DBR is contained in a system wide segment number, the stack segment in each process address space will also have the same number (Segment #1 in this implementation). Each stack segment is initially created as though it's process had been previously preempted by a hardware interrupt. This greatly simplifies the initialization of processes at system generation time. The details of system initialization will be described later in this chapter. It is important to note here, however, that GETWORK must be able to determine whether it was invoked by a hardware preempt interrupt or by a normal call, before it can execute a return to the calling procedure. This is because a hardware interrupt causes three items to be placed on the system stack: the return location of the caller, the flag control word, and the interrupt identifier, whereas a normal call places only the return location on the stack. Therefore, in order to clean up the stack, GETWORK must execute an interrupt return (assembly instruction:IRET) if entry was via the hardware preempt handler (i.e., RØ set). This instruction will pop the three items off the stack and return to the appropriate location. If the interrupt return flag, RØ, is off, a normal return is executed.

During normal operation, SWAP-DBR manipulates process stacks to save the old VP state and load the new VP state. This action proceeds as follows (figure 13):

1. The Flag Control Word (FCW), the Stack Pointer (R15) and the preempt return flag (RØ) are saved in the old VP's kernel stack.

2. The DBR (R14) is loaded with the new VP's DBR. This permits access to the address space of the new process.

3. The Flag Control Word (FCW), the Stack Pointer (R15) and the Interrupt Return Flag (RØ), are loaded into the appropriate CPU registers.

4. RØ is tested. If it is set, GETWORK will execute an interrupt return. If it is off, a normal return occurs.

Kernel Stack Segments

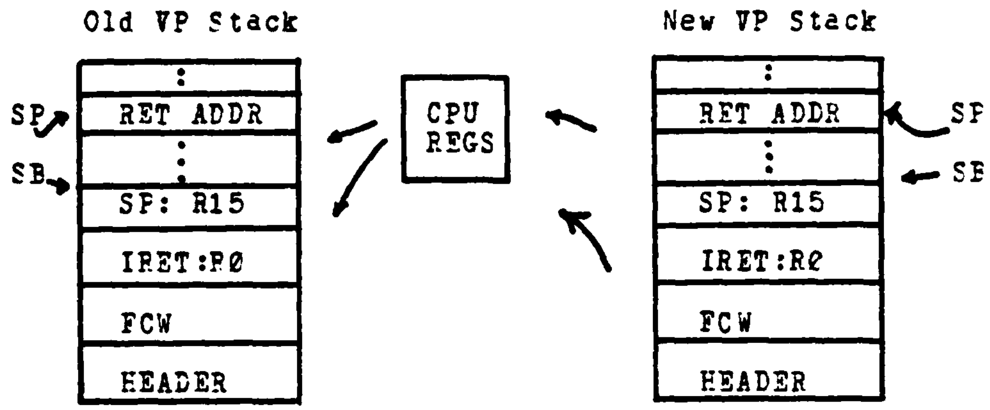


FIGURE 13

By constructing GETWORK in this way, both system initialization and normal operations can be handled in the same way. A high level GETWORK algorithm is given in figure 14.

3. <u>Virtual Processor Instruction Set</u>

The heart of the SASS scheduling mechanism is the internal procedure, GETWORK. It provides a powerful internal primitive for use by the virtual processors and greatly simplifies the design of the virtual processor instruction set. Virtual processor instructions perform three types of functions: multiprogramming, process management and virtual interrupts.

```
GETWORK Procedure (DBR = R14)

 Begin

  Reset Interrupt Return Flag (RØ)

  Skip hardware preempt handler

 Hardware Preempt Entry:
    Set DBR
    Save CPU registers
    Save supervisor stack pointer
    Set Interrupt Return Flag (RØ)

  Get first ready VP

  Do while not Select
   If Idle flag is set then
    if Preempt flag is set then
     select
    else
     get next ready VP
    end if
   else
    select
   end if
  end do

  SWAP_DBR:
   Save old VP registers in stack segment
   Swap dbr (R14)
   Load new VP registers in stack segment

   If Interrupt Return Flag is set then
    unlock VPT

    simulate GATEKEEPER exit:
     Call TEST_VPREEMPT
     Restore supervvisor registers
     Restore supervvisor stack pointer

    Execute Interrupt Return (IRET)
   end if

   Execute normal return

end GETWORK
```

Figure 14

SIGNAL and WAIT provide synchronization and communication between virtual processors. They multiplex virtual processors on a CPU to provide multiprogramming. This implementation used a version of the signal and wait algorithms proposed by Saltzer [10]. In the SASS design each CPU is provided with a unique (fixed) set of virtual processors. The interaction among virtual processors is a result of multiprogramming them on the real processor. Only one virtual processor is able to access the VPT at a time because of the use of the VPT LOCK (SPIN_LOCK) to provide mutual exclusion. Therefore race and deadlock conditions will not develop and the signal pending switch used by Saltzer is not necessary.

This implementation also included message passing mechamism not provided by Saltzer. The message slots available for use by virtual processors are initially contained in a queue pointed to by FREE-LIST. When a message is sent from one VP to another, a message slot is removed from the free list and placed in a FIFO message queue belonging to the VP receiving the message. The head of each VP's message queue is pointed to by MSG-LIST. Each message slot contains a message, the ID of the sender, and a pointer to the next message in the list (either the free list or the VP message list.

IDLE and SWAP_VDBR provide the Traffic Controller with a means of scheduling processes on the running VP.

SET_VPREEMPT and TEST_VPREEMPT install a virtual interrupt mechanism in each virtual processor. When the Traffic Controller determines that a virtual processor should give up its process because a higher priority process is now ready, it sets the PREEMPT flag in that VP. Then, even if an idle process is loaded on the VP, it will be scheduled and will be loaded with the first ready process. Test_VPreempt is a virtual interrupt unmasking mechanism which forces a process to examine the preempt flag each time it exists from the kernel.

a. Wait

WAIT provides a means for a virtual processor to move itself from the running state to the waiting state when it has no more work to do. It is invoked only for system events that are always of short duration. It is supported by three internal Procedures.

SPIN_LOCK enables the running VP to gain control of the Virtual Processor Table. This procedure is only necessary in a multiprocessor environment. The running VP will have to wait only a short amount of time to gain control of the VPT. SPIN_LOCK returns when the VP has locked the VPT.

GETWORK loads the first eligible virtual processor of the ready list on the real processor. Before this procedure is invoked, the running VP is placed in the ready state. Both ready and running VP's are members of a FIFO queue. GETWORK selects the first VP in this ready list. loads it on the CPU, and places it in the running state. When GETWORK returns, the first VP of the queue will always be running and the second will be the first VP in the ready queue.

GET_FIRST_MESSAGE returns the first message of the message list (also managed as a FIFO queue) associated with the running VP. The action taken by WAIT is as follows:

```
WAIT Procedure (Returns: Msg, Sender_ID)

 Begin

  Lock VPT (call SPIN_LOCK)

  If message list empty (i.e., no work) Then
   Move VP from Running to Waiting state
   Schedule first eligible Ready VP (call GETWORK)
  end if

  (NOTE: process suspended here until
         it receives a signal and is
         selected bu GETWORK.)

  Get first message from message list
       (call GET_FIRST_MSG)

  Unlock VPT

  Return

 end WAIT
```

If the running virtual processor calls WAIT and there is a message in its message list (placed there when another VP signaled it) it will get the message and continue to run. If the message list is empty it will place itself in the wait state, schedule the first ready virtual processor, and move it to the running state. The virtual processor will remain in the waiting state until another running VP sends it a message (via SIGNAL). It will then move to the ready list. Finally it will be selected by GETWORK, the next instructions of WAIT will be executed, it will receive the message for which it was waiting, and it will return to the caller.

b. Signal

Messages are passed between virtual processors by the instruction, SIGNAL, which uses four internal procedures, SPIN_LOCK, ENTER_MSG_LIST, MAKE_READY, and GETWORK.

SPIN_LOCK, as explained above insures that only one virtual processor has control of the Virtual Processor Table at a time.

ENTER_MSG_LIST manages a FIFO message queue for each virtual Processor and for free messages. This queue is of fixed maximum length because of the implementation decision to restrict the use of SIGNAL. A running VP can send no more than one message (SIGNAL) before it receives a reply (i.e., WAIT's for a message). Therefore if there are N virtual processors per real processors, the message queue length, L, is:

$$L = N - 1$$

MAKE_READY manages the virtual processor ready queue. If a message is sent to a VP in the waiting state, MAKE_READY wakes it up (it places it in the ready state) and enters it in the ready list. If a running VP signals a waiting VP of higher priority, it will place itself back in the ready state and the higher priority VP will be selected. The action taken by signal is as follows:

```
SIGNAL Procedure (Message, Destination_VP)

 Begin

   Lock VPT (call SPIN_LOCK)

   Send message (call ENTER_MSG_LIST)

   If signaled VP is waiting Then
    Wake it up and make it ready
       (call MAKE_READY)
   end if

   Put running VP in ready state.

   Schedule first elgible ready VP
       (call GETWORK)

   Unlock VPT

   Return (Success_code)

 End SIGNAL
```

c. SWAP_VDBR

SWAP_VDBR contains the same processor switching mechanism used in SWAP_DBR, but applies it to a virtual processor rather than a real processor. Switching is quite simple in this virtual environment because both processor execution point and address space are defined by the Descriptor Base Register. SWAP_VDBR is invoked by the Traffic Controller to load a new process on a virtual processor in support of level-2 scheduling. It uses GETWORK to control the associated level-1 scheduling. The action taken by SWAP_VDBR is:

```
SWAP_VDBR Procedure (New_DBR)
 Begin
  Lock VPT (call SPIN_LOCK)
  Load running VP with New_DBR
  Place running VP in ready state
  Schedule first eligible ready VP
      (call GETWORK)
  Unlock VPT
  Return
 End SWAP_VDBR
```

In this implementation one restriction is placed upon the use of this instruction. If a virtual processor's message list contains at least one message, it can not give up its current DBR. This problem is avoided as the natural result of using SIGNAL and WAIT only for system events, and "masking" preempts within the kernel. If this were permitted, the messages would lose their context. (The messages in a VP_MSG_LIST are actually intended for the process loaded on the VP.)

d. IDLE

The IDLE instruction loads the Idle DBR on the running virtual processor. Only virtual processors in contention for process scheduling will be loaded by this instruction. (The Traffic

Controller is not even aware of virtual processors permanently bound to kernel processes.)

IDLE has the same scheduling effect as SWAP_VDBR, but it also sets the IDLE_FLAG on the scheduled VP. The distinction is made between the two cases because, although the Traffic Controller must schedule an Idle process on the VP if there are no other ready processes, the Inner Traffic Controller does not wish to schedule an Idle VP if there is an alternative. This would be a waste of physical processor resources. The setting of the IDLE_FLAG by the Traffic Controller aids the Inner Traffic Controller in making this scheduling decision. Logically, there is an idle process for each VP; actually the same address space (DBR) is used for all idle processes for the same CPU, since only one will run at a time. As previously explained, virtual processors loaded by this instruction will be selected by GETWORK only to give the Idle process away for a new process in response to a virtual preempt interrupt. The action of IDLE is:

IDLE Procedure

Begin

Lock VPT (call SPIN_LOCK)

Load running VP with Idle DBR

Set VP's IDLE_FLAG

Place running VP in ready state

Schedule first elgible ready VP
(call GETWORK)

Unlock VPT

Return

End IDLE

e. SET_VPREEMPT

SET_VPREEMPT sets the preempt interrupt flag on a specified virtual processor. This forces the virtual processor into level-1 scheduling contention, even if it is loaded with an Idle process. The instruction retrieves an idle virtual processor in the same way a hardware preempt retrieves an idle CPU by forcing the VP to be selected by GETWORK. The only difference between the two cases is the entry point used in GETWORK. The action of SET_VPREEMPT is:

```
SET_VPREEMPT Procedure (VP)
 Begin
  Set VP's PREEMPT flag
  If VP belongs to another CPU Then
   send hardware interrupt
  end if
  Return
 End SET_VPREEMPT
```

Since the action is a safe sequence, no deadlocks or race conditions will arise and no lock is required on the VPT.

f. TEST_VPREEMPT

Within the kernel of a multiprocessor system all process interrupts (which excludes system I/O interrupts) are masked. If process interaction results in a virtual preempt being sent to the running virtual processor by another CPU, it will not be handled since GETWORK has already been invoked. TEST_VPREEMPT provides a virtual preempt interrupt unmasking mechanism.

TEST_VPREEMPT mimics the action of a physical CPU when interrupts are unmasked. It forces the process execution point back down into the kernel each time the process attempts to leave the kernel domain, where the preempt flag of the running VP is examined. If the flag is off, TEST_VPREEMPT returns and the execution point exits through the Gatekeeper into the supervisor domain of the process address space as described above. However, if the PREEMPT flag is on, the TEST_VPREEMPT executes a virtual interrupt handler located in the Traffic Controller. This jump from the Inner Traffic Controller to the Traffic Controller (TC_PREEMPT_HANDLER) is a close parallel to the action of a CPU in response to a hardware interrupt, that is a jump to an interrupt handler. The Traffic Controller Preempt Handler forces level-2 and level-1 scheduling to proceed in the normal manner. The preempt handler forces the Traffic Controller to examine the APT and to apply the level-2 scheduling algorithm, TC_GETWORK. If the APT has been changed since the last invocation of this scheduler, it will be reflected in the scheduling selections. Eventually, when the running VP's preempt flag is tested and found to be reset, TEST_VPREEMPT will return to the Gatekeeper where the process execution point will finally make a normal exit into its supervisor domain. TEST_VPREEMPT performs the following action:

```
TEST_VPREEMPT Procedure

 Begin

  Do while running VP's PREEMPT flag is set
   Reset PREEMPT flag
   Call preempt handler
      (call TC_PREEMPT_HANDLER)
  End do

  Return

 End TEST_VPREEMPT
```

C. TRAFFIC CONTROLLER

The Traffic Controller runs in a virtual environment created by the Inner Traffic Controller. It sees a set of running virtual processor instructions: SWAP_VDBR, IDLE, SET_VPREEMPT, and RUNNING_VP, and provides a scheduler, TC_GETWORK, which multiplexes processes on virtual processors in response to process interaction. It also creates a level-2 instruction set: ADVANCE, AWAIT, and PROCESS_CLASS, which is available for use by higher levels of the design. The Traffic Controller uses a global data base, the ACTIVE PROCESS TABLE to support its operation.

1. <u>Active Process Table (APT)</u>

The Active Process Table is a system-wide kernel database containing entries for each supervvisor process in SASS (Figure 15). It is indexed by active process ID.

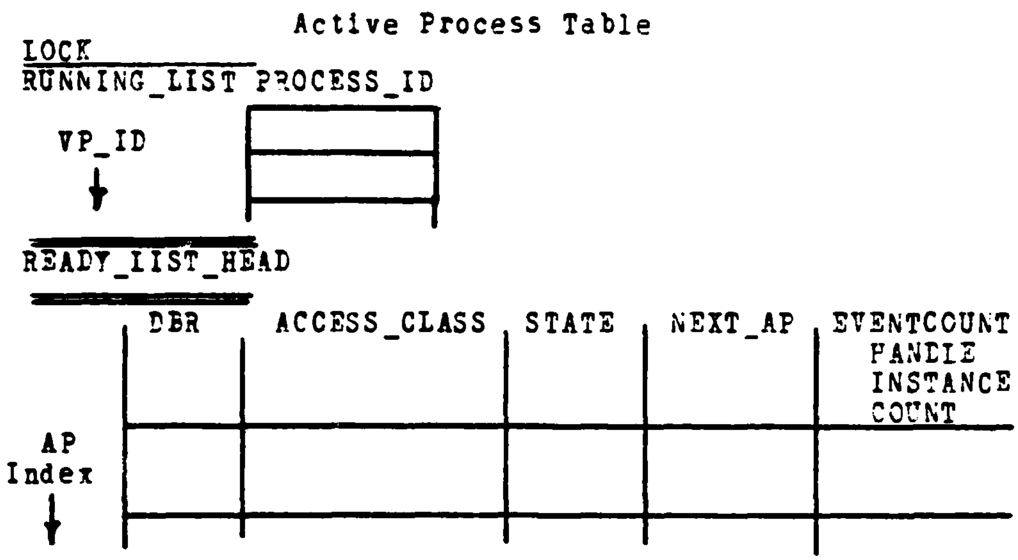


Figure 15

The structure of the APT closely parallels that of the Virtual Processor Table. It contains a LOCK to support the implementation of a mutual exclusion mechanism, a RUNNING_LIST, and a READY_LIST_HEAD. The Traffic Controller is only concerned with virtual processors that can be loaded with supervisor processes. Since two VP's are permanently bound to kernel processes (the Memory Manager and the Idle Process), they cannot be in contention for level-2 scheduling; the Traffic Controller is unaware of their existence; since there are a number of available virtual processors, the RUNNING_LIST was implemented as an array indexed by VP_ID. The READY_LIST_HEAD points to a FIFO queue that includes both running and ready processes. The running processes will be at the top of the ready list.

Because of their completely static nature, idle processes require no entries in the APT. Logically, there is an idle process at the end of the ready list for each VP available to the Traffic Controller. If the ready list is empty, TC_GETWORK loads one of these "virtual" idle processes by calling IDLE, and enters a reserved identifier, #IDLE, in the appropriate RUNNING_LIST entry. This identifier is the only data concerning idle processes that is contained in the APT. Idle process scheduling considerations are moved down to level-1, because the Inner Traffic Controller knows about physical processors, and can optimize CPU use by scheduling idle processes only when there is nothing else to do.

The subject access class, S_CLASS, provides each process with a label that is required by level-3 modules to enforce, the SASS non-discretionary security policy.

2. <u>Level-2 Scheduling</u>

Above the Traffic Controller, SASS appears as a collection of processes in one of the three states: running, ready, or blocked. Running and ready states are analogous to the corresponding virtual processor states of the Inner Traffic Controller. However, because of the use of eventcount synchronization mechanisms by the Traffic Controller, the blocked state has a slightly different connotation than the VP waiting state.

Blocked processes are waiting for the occurrence of a non-system event, e.g., the event occurrence may be signalled from the supervisor domain. When a specific event happens, all of the blocked processes that were awaiting that event are awakened and placed in the ready state. This broadcast feature of event occurrence is more powerful than the message passing mechanism of SIGNAL, which must be directed at a single recipient.

Just as SIGNAL and WAIT provide virtual processor multiplixing in level-1, the eventcount functions, ADVANCE and AWAIT, control process scheduling in level-2.

a. TC_GETWORK

Level-2 scheduling is implemented in the internal Traffic Controller procedure, TC_GETWORK. This procedure is invoked by eventcount functions when a process state change may have occurred. It loads the first ready process on the currently scheduled VP (i.e., the virtual processor that has been scheduled at level-1 and is currently executing on the CPU).

```
TC_GETWORK Procedure

 Begin

  VP_ID := RUNNING_VP

  Do while not end of ready list
   if process is running then
    get next ready process
   else
    RUNNING_LIST [VP_ID] := PROCESS_ID
    Process state := running
    SWAP_VDBR
   end if
  end do

 If end of running list (no ready processes) Then
  RUNNING_LIST := #IDLE
  IDLE
 end if

 Return

End TC_GETWORK
```

A source listing of TC_GETWORK is contained in Appendix B.

b. TC_PREEMPT_HANDLER

Preempt interrupts are masked while a process is executing in the kernel domain. As the process leaves the kernel, the gatekeeper unmasks this virtual interrupt by invoking TEST_VPREEMPT. This instruction tests the scheduled VP's PREEMPT flag. If this flag is off, the process returns to the Gatekeeper and exits from the kernel; but if the flag is set, TEST_VPREEMPT calls the Traffic Controller's virtual preempt interrupt handler, TC_PREEMPT_HANDLER. This handler invokes TC_GETWORK, which re-evaluates level-2 scheduling. Eventually, when the schedulers have completed their functions, the handler will return control to the preempted process, which will return to te Gatekeeper for a normal exit. This sequence of events closely parallels the action of a hardware interrupt, but in the environment of a virtual processor rather than a CPU. The virtualization of interrupts provides the ability for one virtual processor to interrupt execution of another that may, or may not, be running on a CPU at that time. This is provided without disrupting the logical structure of the system. This capability is particularly useful in a multiprocessor environment where the target virtual processor may be executing on another CPU. Because these interrupts will be virtualized, the operating system will retain control of the system. The action of the TC_PREEMPT_HANDLER is described in the procedure below. A source listing is contained in Appendix B.

```
TC_PREEMPT_HANDLER Procedure

 Begin

  Call WAIT_LOCK

  VP_ID := RUNNING_VP

  Process_ID := RUNNING LIST [VP_ID]

  If process is not idle  Then
   Process state := ready
  end if

  Call TC_GETWORK

  Call WAIT_UNLOCK

  RETURN

 End TC_PREEMPT_HANDLER
```

WAIT_LOCK and WAIT_UNLOCK provide an exclusion mechanism which prevents simultaneous multiple use of the APT in a multiprocessor configuration. This mechanism invokes WAIT and SIGNAL of the Inner Traffic Controller.

3. Eventcounts

An eventcount is a non-decreasing integer associated with a global object called an event [11]. The Event Manager, a level-3 module, controls access to event data when required and provides the Traffic Controller with a HANDLE, an INSTANCE, and a COUNT. The values for all eventcounts (and sequencers) are maintained at the Memory Manager level and are accessed by calls to the Memory Manager. The HANDLE provides the traffic controller with an event ID, associated with a particular segment. INSTANCE is a more specific definition of the event. For example, each SASS supervisor segment has two eventcounts associated with it, a INSTANCE_1 and a INSTANCE_2, that the supervisor uses keep track of read and write access to the segment [2]. Eventcounts provide information concerning system-wide events. They are manipulated by the Traffic Controller functions ADVANCE and AWAIT and by the Memory Manager functions, READ and TICKET. A proposed high level design for ADVANCE and AWAIT is provided in Appendix C.

a. Advance

ADVANCE signals the occurrence of an event (e.g., a read access to a particular supervisor segment). The value of the eventcount is the number of ADVANCE operations that have been performed on it. When an event is advanced, the fact must be broadcast to all blocked processes awaiting it and the process must be awakened and placed on the ready list. Some of the newly awakened processes may have a higher priority than some of the running processes. In this case a virtual preempt, SET_VPREEMPT (VP_ID), must be sent to the virtual processors loaded with these lower priority processes.

b. Await

When a process desired to block itself until a particular event occurs, it invokes AWAIT. This procedure returns to the calling process when a specified eventcount is reached. Its function is similar to WAIT.

c. Read

READ returns the current value of the eventcount. This is an Event Manager (level three) function. This module calls the Memory Manager module to obtain the eventcount value.

d. Ticket

TICKET provides a complete time-ordering of possibly concurrent events. It uses a non-decreasing integer, called a sequencer, which is also associated with each supervisor segment. As with READ, this is an Event Manager function that calls the Memory Manager to access the sequencer value. Each invocation of TICKET increments the value of the sequencer and returns it to the caller. Two different uses of ticket will return two different values, corresponding to the order in which the calls were made.

D. SYSTEM INITIALIZATION

Because the Inner Traffic Controller's scheduler, GETWORK, can accommodate both normal calls and hardware interrupt jumps, the problem of system initialization is not difficult.

When SASS is first started at level-1, the Idle VP is running and the memory manager VP, which has the highest priority, is the first ready virtual processor in the ready list. All VP's available to the Traffic Controller for level-2 schedling are ready. Their IDLE_FLAG's and PREEMPT flags are set.

At level-2, all VP's are loaded with idle processes and all supervisor processes are ready.

The kernel stack segment of each process is initialized to appear as if it had been saved by a hardware Preempt interrupt (Figure 16).

Initialized Stack

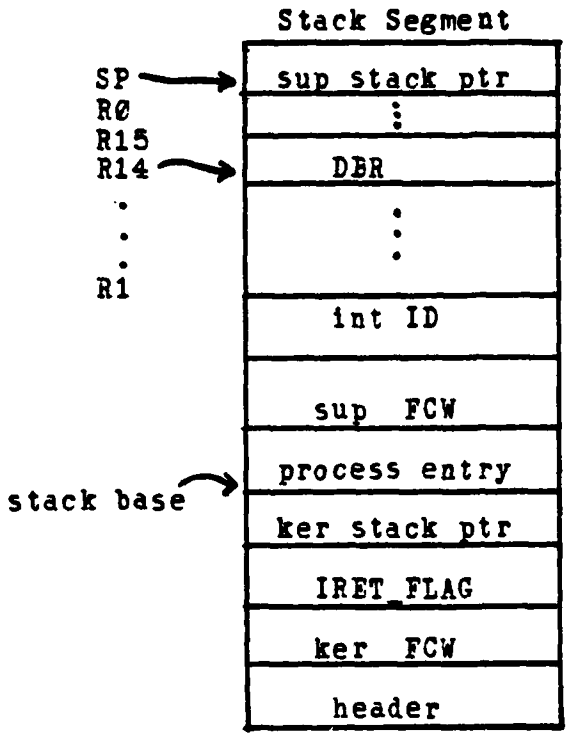


Figure 16

All CPU registers and the supervisor stack pointer are stored on the stack. R15 is reserved as the kernel stack point; R14 contains the DBR. All other registers can be used to pass initial parameters to the process. The order in which these registers appear on the stack supports the Z/ASM block-move instructions.

The status block contains the current value of the stack pointer, R15, and the preempt interrupt return flag. This flag is set to indicate that the process has been saved by a preempt interrupt. The first three items on the stack: the process entry point, the initial process flag control word, and an interrupt indentifier, are also initialized to support the action of a hardware interrupt.

To start-up the system, R14 (the DBR) is set to the Idle process DBR; the CPU Program counter is assigned the PREEMPT_ENTRY point in GETWORK; the CPU Flag Control Word (FCW) is initialized for the kernel domain; and the CPU is started. Because the Idle_VP is the lowest priority VP in the system, it will place itself back in the ready state and move the Memory Manager in the running state. The Memory Manager will execute an interrupt return because the interrupt return flag was set by system initialization. There will be no Work for this kernel process so it will call WAIT to place itself in the waiting state. The next ready VP is idling, but since it's IDLE_FLAG and PREEMPT flag are set, GETWORK will select it. It too will execute an interrupt return, but because its PREEMPT flag is set, it will call TC_PREEMPT_HANDLER. This will cause the first ready process to be scheduled. Each time a supervisor process blocks itself, the next idle VP will be selected and the sequence will be repeated.

The action described above is in accord with normal operation of the system. The only unique features of initialization are the entry point (PREEMPT-ENTRY: in GETWORK) and the values in the initialized kernel stack.

The implementation presented in this thesis has been run on a Z8000 developmental module. System initialization has been tested and executes correctly. At the current level of implementation, no process multiplexing function is available. There is no provision for unlocking the APT after an initialized process has been loaded as a result, a call to the Traffic Contorller (viz., ADVANCE or AWAIT). In a process multiplexed environment this would cause a system deadlock. Once the process left the kernel domain with a locked APT, no process would be able to unlock it. The Traffic Controller must handle this system initialization problem.

V. CONCLUSION

The implementation presented in this thesis created a security kernel monitor that runs on the Z8000 Developmental Module. This monitor supports multiprogramming and process management in a distributed operating system. The process executes in a multiple virtual processor environment which is independent of the CPU configuration.

This monitor was designed specifically to support the Secure Archival Storage System (SASS) [1, 2, 3]. However, the implementation is based on a family of Operating Systems [4] designed with a primary goal of providing multilevel security of information. Although the monitor currently runs on a single microprocessor system, the implementation fully supports a multiprocessor design.

A. RECOMMENDATIONS

Because the Zilog MMU is not yet available for the Z8000 Developmental Module, it was necesary to simulate the segmentation hardware. As explained in Chapter IV, this was accomplished by reserving a CPU register, R14, as a Descriptor Base Register (DBR) to provide a link to the loaded addresss space. When the MMU becomes available, this simulation must be removed. This can be done in two steps.

First, the addressing format must be translated to the segmented form. This requires no system redesign.

Second, the switching mechanism most be modified to accomodated to use the MMU. This can be done by modifying the SWAP_DBR portion of GETWORK to multiplex the MMU_IMAGE onto the MMU hardware and this can be accomplished by changing about a dozen lines of the existing code.

B. FOLLOW ON WORK

Although the monitor appears to execute correctly, it has not been rigorously tested. Before higher levels of the system are added, it is essential that the monitor be highly reliable. Therefore a formal test and evaluation plan should be developed.

An automated system generation and initialization mechanism is also required if the monitor to be is a useful tool in the development of higher levels of the design.

Once the monitor has been proven reliable and can be loaded easily, work on the implementation of the Memory Manager kernel process and the remainder of the kernel can continue.

```
Z8000ASM  2.02
LOC    OBJ CODE    STMT SOURCE STATEMENT

INNER_TRAFFIC_CONTROL MODULE

! * * * VERS. 1.4 * * * !
      ! ** NOTE:
   1. GETWORK:
     A. NORMAL ENTRY DOES NOT SAVE ANY REGS.
        ( THIS FUNCTION OF GATEKEEPER ).
     B. R14 IS INPUT PARAMETER SUMULATING INFO WHICH
        WILL EVENTUALLY BE AVAILABLE ON THE HARDWARE (MMU).
        THIS REG IS ESTABLISHED AS THE DBR BY ANY ITC
        PROCEDURE CALLING GETWORK AND BY THE PREEMPT
        INTERRUPT HANDLER (PREEMPT_ENTRY).
   2. GENERAL:
     A. ALL VIOLATIONS OF VIRTUAL
        MACHINE INSTRUCTIONS ARE CONSIDERED ERROR CONDITIONS
        AND WILL CRASH SYSTEM (RETURNING AN ERROR_CODE: RØ).
     B. ALL ITC PROCEDURES CALLING GETWORK PASS DBR: R14
        AS INPUT PARAMETER.(SIG, WAIT, SWAP_VDBR, & IDLE) **
 CONSTANT
   ! ********** ERROR CODES ********** !
        UNAUTH_LOCK             := Ø
        MSG_LIST_EMPTY          := 1
        MSG_LIST_ERROR          := 2
        READY_LIST_EMPTY        := 3
        MSG_LIST_OVERFLOW       := 4
        SWAP_NOT_ALLOWED        := 5 ! MSG_LIST_NOT EMPTY !
        VP_INDEX_ERROR          := 6
```

```
        ! ******** SYSTEM PARAMETERS ******** !
          NR_MMU_REG       := 64   !LONG WORDS!
          NR_VP            := 4
          IDLE_VP          := NR_VP-1
          STACK_SEG        := 1
          STACK_SEG_SIZE   := %100
            ! * * OFFSETS IN STACK SEG * * !
          STACK_BASE       := STACK_SEG_SIZE-%40
          STATUS_REG_BLOCK:= STACK_SEG_SIZE-%40
          F_C_W            := STACK_SEG_SIZE-%20
          PROCESS_ID       := STACK_SEG_SIZE-%1E
          N_S_P            := STACK_SEG_SIZE-%1C

          ON      := %FFFF
          OFF     := 0
          RUNNING := 0
          READY   := 1
          WAITING := 2
          NIL     := %FFFF
          INVALID := %EEEE
          MONITOR  := %A900       ! HBUG ENTRY !
          TC_PREEMPT_HANDLER := %A828
 !PAGE
```

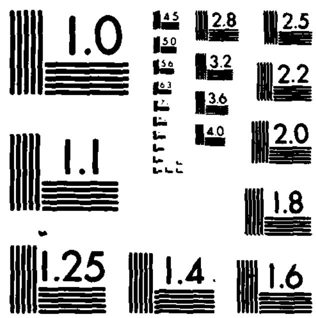
```
TYPE
 MESSAGE          WORD
 ADDRESS          WORD
 VP_INDEX         INTEGER
 MSG_INDEX        INTEGER

 MMU_TABLE RECORD [ BASE          ADDRESS
                    ATTRIBUTES    WORD
                  ]

MSG_TABLE RECORD [ MSG            MESSAGE
                   SENDER         VP_INDEX
                   NEXT_MSG       MSG_INDEX
                   FILLER         ARRAY [5, WORD]
                 ]

VP_TABLE RECORD       [ DBR               ADDRESS
                        PRI               WORD
                        STATE             WORD
                        IDLE_FLAG         WORD
                        PREEMPT           WORD
                        PHYS_PROCESSOR    WORD
                        NEXT_READY_VP     VP_INDEX
                        MSG_LIST          MSG_INDEX
                        FILLER_1          ARRAY [8, WORD]
                ]
```

AD-A091 092 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
AN IMPLEMENTATION OF MULTIPROGRAMMING AND PROCESS MANAGEMENT FO--ETC(U)
JUN 80 S L REITZ

UNCLASSIFIED NL

2 of 2

END
DATE
FILMED
12-80
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
INTERNAL
$SECTION DATA

0000      VPT  RECORD
               [ LOCK          WORD
                 RUNNING_LIST VP_INDEX
                 READY_LIST   VP_INDEX
                 FREE_LIST    MSG_INDEX
                 FILLER_2     ARRAY [4, WORD]
                 VP           ARRAY [NR_VP, VP_TABLE]
                 MSG_Q        ARRAY [NR_VP, MSG_TABLE]
               ]
!PAGE
```

```
!
                    $SECTION INT_PROC
0000                  GETWORK         PROCEDURE
                                      !*****************************!
                                      ! SWAPS VIRTUAL PROCESSORS    !
                                      ! ON PHYSICAL PROCESSOR.      !
                                      !*****************************!
                                      ! REGISTER USE:               !
                                      !  STATUS REGISTERS           !
                                      !   RØ: INTERRUPT_RETURN_FLAG !
                                      !   R14: DBR (SIMULATION)     !
                                      !   R15: STACK_POINTER        !
                                      !  LOCAL VARIABLES:           !
                                      !   R1: READY_VP (NEW)        !
                                      !   R2: CURRENT_VP (OLD)      !
                                      !   R3: FLAG CONTROL WORD     !
                                      !   R4: STACK_SEG BASE ADDR   !
                                      !   R5: STATUS_REG_BLOCK ADDR !
                                      !   R6: NORMAL STACK POINTER  !
                                      !*****************************!
                        ENTRY
                             ! TURN OFF PREEMPT_RETURN_FLAG !
ØØØØ 21ØØ  ØØØØ              LD      RØ, #OFF

                             ! GET STACK BASE !
ØØØ4 31E4  ØØØ4              LD      R4, R14(#STACK_SEG*4)
ØØØ8 3445  ØØCØ              LDA     R5, R4(#STATUS_REG_BLOCK)

                             ! SKIP PREEMPT_HANDLER !
ØØØC E817                    JR      END_PREEMPT_HANDLER
```

```
                          PREEMPT_ENTRY:      ! GLOBAL LABEL !

                             ! * * PREEMPT_HANDLER * * !

                                ! SET DBR !
000E 6102  0002'                LD   R2, VPT.RUNNING_LIST
0012 612E  0010'                LD   R14, VPT.VP.DBR(R2)

                                ! PUT CURRENT PROCESS IN READY STATE !
0016 4D25  0014'                LD   VPT.VP.STATE(R2), #READY
001A 0001

                                ! SAVE ALL REGISTERS !
001C 030F  0020                 SUB  R15, #32
0020 1CF9  010F                 LDM  @R15, R1, #16

                                ! SAVE NORMAL STACK POINTER (NSP) !
0024 7D67                       LDCTL R6, NSP
0026 93F6                       PUSH @R15, R6

                                ! SAVE LAST STATUS_REGS !
                             ! NOTE: SINCE PROCESSES CAN BE PREEMPTED ANYWHERE
                                IT IS NECESSARY TO HANDLE RECURSIVE CALLS
                                TO GETWORK. BY SAVING THE MOST RECENT SP
                                AND IRET_FLAGS (R15 & R0) ON THE STACK
                                THE CONTEXT OF THESE STATUS REGISTERS IS
                                MAINTAINED TO ANY DEPTH OF RECURSION. !
```

```
                          ! SAVE LAST STATUSS_REGS !
0030 1C51  0701           LDM  R7, @R5, #2
0034 93F7                 PUSH @R15, R7
0036 93F8                 PUSH @R15, R8

                          ! SET INTERRUPT RETURN FLAG !
0038 2100  FFFF           LD   R0, #ON
                        ! * * * * * * * * * * * * * * !


                        END_PREEMPT_HANDLER:

                        ! GET READY_VP LIST !
003C 6101  0004'        LD        R1, VPT.READY_LIST

                        SELECT_VP:
                        DO  ! UNTIL ELGIBLE READY_VP FOUND !

0040 4D11  0016'         CP       VPT.VP.IDLE_FLAG(R1), #ON
0044 FFFF
0046 5E0E  005C'         IF EQ  ! VP IS IDLE ! THEN
004A 4D11  0018'          CP VPT.VP.PREEMPT(R1), #ON
004E FFFF
0050 5E0E  0058'          IF EQ  ! PREEMPT INTERRUPT IS ON !  THEN
0054 5E08  0068'           EXIT FROM SELECT_VP
                          FI
0058 5E08  0060'         ELSE ! VP NOT IDLE !
005C 5E08  0068'          EXIT FROM SELECT_VP
                         FI

                         ! GET NEXT READY_VP !
0060 6113  001C'         LD       R3, VPT.VP.NEXT_READY_VP(R1)
0064 A131                LD       R1, R3
0066 E8EC               OD
                   !PAGE
```

```
                    ! NOTE: THE READY_LIST WILL NEVER BE EMPTY SINCE
                            THE IDLE VP, WHICH IS THE LOWEST PRI VP,
                            WILL NEVER BE REMOVED FROM THE LIST.
                      IT WILL RUN ONLY IF ALL OTHER READY VP'S ARE
                      IDLING OR IF THERE ARE NO OTHER VP'S ON
                      THE READY_LIST. ONCE SCHEDULED, IT
                      WILL RUN UNTIL RECEIVING A HARDWARE INTERRUPT. !

                SWAP_DBR:
                      ! * * SAVE SP AND INTERRUPT RETURN FLAG * * !
                ! NOTE: R14 IS USED AS DBR HERE. WHEN MMU HARDWARE
                      IS AVAILABLE THIS SERIES OF SAVE AND LOAD
                      INSTRUCTIONS WILL BE REPLACED BY SPECIAL I/O
                      INSTRUCTIONS TO THE MMU. !

0068 1C59  0F01       LDM      @R5, R15, #2

                      ! * * SAVE FCW * * !
006C 7D32             LDCTL    R3, FCW
006E 3343  00E0       LD       R4(#F_C_W), R3

                      ! PLACE NEW_VP IN RUNNING STATE !
0072 4D15  0014'      LD       VPT.VP.STATE(R1), #RUNNING
0076 0000
0078 6F01  0002'      LD       VPT.RUNNING_LIST, R1

                      ! SWAP DBR !
007C 611E  0010'      LD       R14, VPT.VP.DBR(R1)

                      ! LOAD NEW_VP SP & INTERRUPT RET FLAG !
0080 31E4  0004       LD       R4, R14(#STACK_SEG*4)
0084 3445  00C0       LDA      R5, R4(#STATUS_REG_BLOCK)
0088 1C51  0F01       LDM      R15, @R5, #2
```

```
                        ! * * LOAD NEW FCW * * !
008C 3143   00E0        LD      R3, R4(#F_C_W)
0090 7D3A               LDCTL   FCW, R3

                        ! TEST FOR HARDWARE INTERRUPT !
0092 0B00   FFFF        CP      R0, #ON
0096 5E0E   00BC'       IF EQ  ! PREEMPT RETURN ! THEN
                         ! HARDWARE PREEMPT INTERRUPT RETURN !

                         ! UNLOCK VPT !
009A 4D08   0000'        CLR  VPT.LOCK

                         ! TEST FOR PREEMPT !
                        ! NOTE: SINCE A HARDWARE INTERRUPT DOES NOT EXIT THE
                                THROUGH THE GATE, THOSE FUNCTIONS PROVIDED
                                BY A GATE EXIT TO HANDLE PREEMPTS MUST BE
                                PROVIDED HERE ALSO. !
009E 5F00   0102'        CALL TEST_PREEMPT

                         ! RESTORE LAST STATUS_REGS !
00A2 97F8                POP    R8, @R15
00A4 97F7                POP    R7, @R15
00A6 1C59   0701         LDM    @R5, R7, #2

                         ! RESTORE NSP !
00AA 97F6                POP R6, @R15
00AC 7D6F                LDCTL   NSP, R6
!
```

```
!
                              ! RESTORE ALL REGSTERS !
00AE 1CF1  010F                LDM    R1, @R15, #16
00B2 010F  0020                ADD    R15, #32

                               ! EXECUTE HARDWARE INTERRUPT RETURN !
00B6 7B00                      IRET

00B8 5E08  00BE'              ELSE  ! NORMAL RETURN !
00BC 9E08                      RET
                              FI
00FE                        END GETWORK
          !PAGE
```

```
00BE                    ENTER_MSG_LIST  PROCEDURE
                                                    !********************************!
                                                    ! INSERTS POINTER TO MESSAGE     !
                                                    ! FROM CURRENT_VP TO SIGNALED_VP!
                                                    ! IN FIFO MSG_LIST               !
                                                    !********************************!
                                                    ! REGISTER USE:                  !
                                                    !  PARAMETERS:                   !
                                                    !   R0:MSG (INPUT)               !
                                                    !   R1: SIGNALED_VP (INPUT)      !
                                                    !  LOCAL VARIABLES:              !
                                                    !   R2: CURRENT_VP               !
                                                    !   R3: FIRST_FREE_MSG           !
                                                    !   R4: NEXT_FREE_MSG            !
                                                    !   R5: NEXT_Q_MSG               !
                                                    !   R6: PRESENT_Q_MSG            !
                                                    !********************************!
                          ENTRY
00BE 6102  0002'             LD      R2, VPT.RUNNING_LIST

                             ! GET FIRST MSG FROM FREE_LIST !
00C2 6103  0006'             LD      R3, VPT.FREE_LIST

                                                    ! * * * * DEBUG * * * * !
00C6 0B03  FFFF                                     CP R3, #NIL
00CA 5E0E  00DA'                                    IF EQ THEN
00CE 7601  00CE'                                     LDA R1, $
00D2 2100  0004                                      LD R0, #MSG_LIST_OVERFLOW
00D6 5F00  A900                                      CALL MONITOR
                                                    FI
                                                    ! * * * END DEBUG * * * !

00DA 6134  0094'             LD      R4, VPT.MSG_Q.NEXT_MSG(R3)
00DE 6F04  0006'             LD      VPT.FREE LIST, R4
```

```
                              ! INSERT MESSAGE LIST INFORMATION !
00E2 6F30  0090'              LD      VPT.MSG_Q.MSG(R3), R0
00E6 6F32  0092'              LD      VPT.MSG_Q.SENDER(R3), R2

                              ! INSERT MSG IN MSG_LIST !
00EA 6115  001E'              LD      R5, VPT.VP.MSG_LIST(R1)

00EE 0B05  FFFF               CP      R5, #NIL
00F2 5E0E  00FE'              IF EQ  ! MSG LIST IS EMPTY !  THEN
                               ! INSERT MSG AT TOP OF LIST !
00F6 6F13  001E'               LD     VPT.VP.MSG_LIST(R1), R3

00FA 5E08  0116'              ELSE  ! INSERT MSG IN LIST !
                               MSG_Q_SEARCH:
                               DO  ! WHILE NOT END OF LIST !

00FE 0B05  FFFF                 CP    R5, #NIL
0102 5E0E  010A'                IF EQ  ! END OF LIST !  THEN
0106 5E08  0112'                 EXIT FROM MSG_Q_SEARCH
                                FI

                                ! GET NEXT LINK !
010A A156                       LD    R6, R5
010C 6165  0094'                LD    R5, VPT.MSG_Q.NEXT_MSG(R6)
0110 E8F6                      OD

                              ! INSERT MSG IN LIST !
0112 6F63  0094'               LD     VPT.MSG_Q.NEXT_MSG(R6), R3

                              FI

0116 6F35  0094'              LD      VPT.MSG_Q.NEXT_MSG(R3), R5

011A 9E08                     RET
011C                       END ENTER_MSG_LIST
```

```
!

011C                    GET_FIRST_MSG       PROCEDURE
                                            !*******************************!
                                            ! REMOVES MSG FROM MSG_LIST     !
                                            ! AND PLACES ON FREE LIST.      !
                                            ! RETURNS SENDER'S MSG AND      !
                                            ! VP_ID                         !
                                            !*******************************!
                                            !REGISTER USE:                  !
                                            ! PARAMETERS:                   !
                                            !  R0: MSG (RETURNED)           !
                                            !  R1: SENDER VP (RETURNED)     !
                                            ! LOCAL VARIABLES               !
                                            !  R2: CURRENT_VP               !
                                            !  R3: FIRST_MSG                !
                                            !  R4: NEXT_MSG                 !
                                            !  R5: NEXT_FREE_MSG            !
                                            !  R6: PRESENT_FREE_MSG         !
                                            !*******************************!
                         ENTRY
011C 6102  0002'            LD      R2, VPT.RUNNING_LIST

                            ! REMOVE FIRST MSG FROM MSG_LIST !
0120 6123  001E'            LD      R3, VPT.VP.MSG_LIST(R2)

                                                    ! * * * * DEBUG * * * * !
0124 0B03  FFFF                                     CP R3, #NIL
0128 5E0E  0138'                                    IF EQ THEN
012C 2100  0001                                      LD R0, #MSG_LIST_EMPTY
0130 7601  0130'                                     LDA R1, $
0134 5F00  A900                                      CALL MONITOR
                                                    FI
                                                    ! * * * END DEBUG * * * !
```

```
0138 6134  0094'         LD      R4, VPT.MSG_Q.NEXT_MSG(R3)
013C 6F24  001E'         LD      VPT.VP.MSG_LIST(R2), R4

                         ! INSERT MESSAGE IN FREE_LIST !
0140 6105  0006'         LD      R5, VPT.FREE_LIST

0144 0B05  FFFF          CP      R5, #NIL
0148 5E0E  015A'         IF EQ   ! FREE_LIST IS EMPTY !  THEN
                          ! INSERT AT TOP OF LIST !
014C 6F03  0006'          LD     VPT.FREE_LIST, R3
0150 4D35  0094'          LD     VPT.MSG_Q.NEXT_MSG(R3), #NIL
0154 FFFF

0156 5E08  0176'         ELSE  ! INSERT IN LIST !

                          FREE_Q_SEARCH:
                          DO

015A 0B05  FFFF            CP    R5, #NIL
015E 5E0E  0166'           IF EQ  ! END OF LIST !  THEN
0162 5E08  016E'            EXIT FROM FREE_Q_SEARCH
                           FI

                           ! GET NEXT MSG !
0166 A156                  LD    R6, R5
0168 6165  0094'           LD    R5, VPT.MSG_Q.NEXT_MSG(R6)
016C E8F6                 OD
                ! PAGE
```

```
!


                          ! INSERT IN LIST !
016E 6F63  0094'           LD      VPT.MSG_Q.NEXT_MSG(R6), R3
0172 6F35  0094'           LD      VPT.MSG_Q.NEXT_MSG(R3), R5
                          FI

                   ! GET MSG INFORMATION ( RETURNS RØ: MSG, R1: SENDING_VP) !
0176 6131  0092'          LD      R1, VPT.MSG_Q.SENDER(R3)
017A 6130  0090'          LD      RØ, VPT.MSG_Q.MSG(R3)

017E 9E08                 RET
0180                  END GET_FIRST_MSG



                   !PAGE
```

```
!
0180                       MAKE_READY     PROCEDURE
                                          !******************************!
                                          ! INSERTS SCHEDULE VP ID INTO  !
                                          ! RFADY LIST IAW PRIORITY AND  !
                                          ! PUTS IT IN READY STATE.      !
                                          !******************************!
                                          ! REGISTER USE:                !
                                          !  PARAMETERS:                 !
                                          !     R1: SIGNALED_VP (INPUT)  !
                                          !   LOCAL VARIABLES            !
                                          !     R2: SIG_VP.PRI           !
                                          !     R3: PRESENT_VP           !
                                          !     R4: NEXT_VP              !
                                          !******************************!
                                ENTRY
0180 6104  0002'                   LD     R4, VPT.RUNNING_LIST

                                                  ! * * * DEBUG * * * !
0184 0B04  FFFF                                   CP      R4, #NIL
0188 5E0E  0198'                                  IF EQ  ! LIST IS EMPTY !  THEN
018C 2100  0003                                    LD  R0, #READY_LIST_EMPTY
0190 7601  0190'                                   LDA  R1, $
0194 5F00  A900                                    CALL MONITOR
                                                  FI
                                                  ! * * END DEBUG * * !
```

```
0198 6112  0012'        LD      R2, VPT.VP.PRI (R1)

019C 4B42  0012'        CP      R2, VPT.VP.PRI(R4)
01A0 5E02  01B0'        IF GT  ! SIG_VP.PRI > READY_VP.PRI !  THEN
                         ! INSERT AT FRONT OF LIST !
01A4 6F14  001C'         LD     VPT.VP.NEXT_READY_VP(R1), R4
01A8 6F01  0004'         LD     VPT.READY_LIST, R1

01AC 5E08  01D8'        ELSE  ! INSERT IN LIST !

                         READY_LIST_SEARCH:
                          DO  ! WHILE NOT END OF LIST !

01B0 0B04  FFFF             CP   R4, #NIL
01B4 5E0E  01BC'            IF EQ  ! IF END OF LIST  !  THEN
01B8 5E08  01D0'             EXIT FROM READY_LIST_SEARCH
                            FI

01BC 4B42  0012'            CP   R2, VPT.VP.PRI (R4)
01C0 5E02  01C8'            IF  GT  !  SIG_VP.PRI > PRESENT_VP.PRI !  THEN
01C4 5E08  01D0'             EXIT FROM READY_LIST_SEARCH
                            FI

                            ! GET NEXT LINK !
01C8 A143                   LD   R3,R4
01CA 6134  001C'            LD   R4, VPT.VP.NEXT_READY_VP(R3)
01CE E8F0               OD

                  !PAGE
```

```
!

                                !   INSERT SIG_VP IN LIST !
01D0 6F14  001C'                LD      VPT.VP.NEXT_READY_VP(R1),  R4
01D4 6F31  001C'                LD      VPT.VP.NEXT_READY_VP(R3), R1

                                FI

                                ! CHANGE STATE TO READY !
01D8 4D15  0014'                LD      VPT.VP.STATE(R1), #READY
01DC 0001

01DE 9E08                       RET


01E0                        END MAKE_READY
                         !PAGE
```

```
                   ! * * * INNER TRAFFIC CONTROL ENTRY POINTS * * * !
                   GLOBAL
                   $SECTION GLB_PROC

                   HARDWARE_PREEMPT LABEL

0000               WAIT    PROCEDURE
                           !**********************************!
                           ! INTRA_KERNEL SYNC/COM PRIMATIVE !
                           ! INVOKED BY KERNEL PROCESSES     !
                           !**********************************!
                           ! PARAMETERS                      !
                           !  R0: SIGNALED_MSG (RETURN)      !
                           !  R1: SENDING_VP (RETURN)        !
                           ! GLOBAL VARIABLES                !
                           !  R14: DBR (PARAM TO GETWORK)    !
                           ! LOCAL VARIABLES                 !
                           !  R2: CURRENT_VP (RUNNING)       !
                           !  R3: NEXT_READY_VP              !
                           !  R4: LOCK_ADDRESS               !
                           !**********************************!
                       ENTRY
                           ! LOCK VPT !
0000 7604  0000'           LDA     R4, VPT.LOCK
0004 5F00  0150'           CALL    SPIN_LOCK  ! (R4:^VPT.LOCK) !
                           ! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP. !
```

```
0008 6102  0002'          LD      R2, VPT.RUNNING_LIST
000C 6123  001C'          LD      R3, VPT.VP.NEXT_READY_VP(R2)

0010 4D21  001E'          CP      VPT.VP.MSG_LIST(R2), #NIL
0014 FFFF
0016 5E0E  0046'          IF EQ  ! CURRENT VP'S MSG LIST IS EMPTY !  THEN

                           ! REMOVE CURRENT_VP FROM READY_LIST !

                                          ! * * * * DEBUG * * * * !
001A 0B03  FFFF                           CP      R3, #NIL
001E 5E0E  002E'                          IF EQ  THEN
0022 2100  0003                            LD  R0, #READY_LIST_EMPTY
0026 7601  0026'                           LDA  R1, $
002A 5F00  A900                            CALL MONITOR
                                          FI
                                          ! * * * END DEBUG * * * !

002E 6F03  0004'              LD  VPT.READY_LIST, R3
0032 4D25  001C'              LD  VPT.VP.NEXT_READY_VP(R2), #NIL
0036 FFFF

                              ! PUT IT IN WAITING STATE !
0038 4D25  0014'              LD VPT.VP.STATE(R2), #WAITING
003C 0002

                              ! SET DBR !
003E 612E  0010'              LD  R14, VPT.VP.DBR(R2)

                              ! SCHEDULE FIRST ELGIBLE READY VP !
0042 5F00  0000'              CALL GETWORK  !(R14: DBR) !
                          FI
                 !PAGE
```

```
!
                         ! GET FIRST MSG ON CURRENT (MAYBE NEW) VP'S MSG LIST !
0046 5F00  011C'             CALL GET_FIRST_MSG   ! RETURNS R0:MSG, R1:SENDER_VP !

                                 ! UNLOCK VPT !
004A 4D08  0000'                 CLR  VPT.LOCK

                                 ! RETURN: R0:MSG, R1:SENDER_VP !
004E 9E08                        RET
0050                         END WAIT

                         !PAGE
```

```
!
0050                  SIGNAL     PROCEDURE
                                 |*********************************|
                                 ! INTRA_KERNEL SYNC /COM PRIMATIVE !
                                 ! INVOKED BY KERNEL PROCESSES      !
                                 |*********************************|
                                 ! REGISTER USE:                    !
                                 !  PARAMETERS:                     !
                                 !   RØ: MESSAGE (INPUT)            !
                                 !   R1: SIGNALED VP_ID (INPUT)     !
                                 ! GLOBAL VARIABLES                 !
                                 !  R14: DBR (PARAM TO GETWORK)     !
                                 !  LOCAL VARIABLES:                !
                                 !   R1: SIGNALED VP                !
                                 !   R2: CURRENT_VP                 !
                                 !   R4: VPT.LOCK ADDRESS           !
                                 |*********************************|
                      ENTRY
                         ! LOCK VPT !
0050 7604  0000'         LDA     R4, VPT.LOCK
0054 5F00  0150'         CALL    SPIN_LOCK  ! (R4:^VPT.LOCK) !
                         ! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP. !

                         ! PLACE MSG IN SIGNALED_VP'S MSG_LIST !
0058 5F00  00BE'         CALL ENTER_MSG_LIST  ! (RØ:MSG, R1:SIGNALED_VP) !

005C 4D11  0014'         CP      VPT.VP.STATE(R1), #WAITING
0060 0002
0062 5E0E  007C'         IF EQ  ! SIGNALED_VP IS WAITING !  THEN

                          ! WAKE IT UP AND MAKE IT READY !
0066 5F00  0180'          CALL MAKE_READY  ! (R1: SIGNALED VP)) !
```

```
                        ! PUT CURRENT_VP IN READY_STATE !
006A 6102  00C2'        LD      R2, VPT.RUNNING_LIST
006E 4D25  0014'        LD      VPT.VP.STATE(R2), #READY
0072 0001

                        ! SET DBR !
0074 612E  0010'        LD      R14, VPT.VP.DBR(R2)

                       ! SCHEDULE FIRST ELGIBLE READY VP !
0078 5F00  0000'        CALL GETWORK  !(R14: DBR) !
                       FI

                       ! UNLOCK VPT !
007C 4D08  0000'       CLR  VPT.LOCK

0080 9E08              RET
0082               END SIGNAL
             !PAGE
```

```
!
0082                     SET_PREEMPT          PROCEDURE
                                              !***************************!
                                              ! SETS PREEMPT INTERRUPT ON!
                                              ! TARGET_VP. CALLED BY TC_  !
                                              ! ADVANCE.                  !
                                              !***************************!
                                              ! REGISTER USE:             !
                                              ! PARAMETERS:               !
                                              !  R1:TARGET_VP_ID          !
                                              ! LOCAL VARIABLES           !
                                              !  R1: VP_INDEX             !
                                              !***************************!
                         ENTRY
                            ! NOTE: DESIGNED AS SAFE SEQUENCE SO VPT NEED NOT
                                    BE LOCKED. !

                            ! CONVERT VP_ID TO VP_INDEX !
0082 BD00                   LDK     R0, #0
0084 1900  0020             MULT    RR0, #SIZEOF VP_TABLE
                              ! THIS LEAVES VP_INDEX IN R1 !

                            ! TURN ON TGT_VP PREEMPT FLAG !
0088 4D15  0018'            LD      VPT.VP.PREEMPT(R1), #ON
008C FFFF

                    ! ** IF TARGET VP NOT LOCAL (NOT CONNECTED TO THIS CPU),
                    [ IE. IF <<PROC_SEG>>PROC_ID <> VPT.VP.PHYS_PROC(R1) ]
                            THEN SEND HARDWARE PREEMPT INTERRUPT TO CPU  ** !

008E 9E08                   RET
0090                     END SET_PREEMPT
                    !PAGE
```

```
!
0090                      IDLE      PROCEDURE
                                    !***********************!
                                    ! LOADS IDLE DBR ON     !
                                    ! CURRENT VP. CALLED BY !
                                    ! TC_GETWORK.           !
                                    !***********************!
                                    ! REGISTER USE          !
                                    !  GLOBAL VARIABLE      !
                                    !   R14: DBR            !
                                    !  LOCAL VARIABLES:     !
                                    !   R2: CURRENT_VP      !
                                    !   R3: TEMP VAR        !
                                    !   R4: VPT.LOCK ADDR   !
                                    !   R5: TEMP            !
                                    !***********************!
                          ENTRY
                            ! LOCK VPT !
0090 7604  0000'            LDA     R4, VPT.LOCK
0094 5F00  0150'            CALL    SPIN_LOCK  ! (R4:^VPT.LOCK) !
                            ! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP. !

                            ! GET CURRENT_VP !
0098 6102  0002'            LD      R2, VPT.RUNNING_LIST

                            ! SET DBR !
009C 612E  0010'            LD      R14, VPT.VP.DBR(R2)
```

```
00A0 2103  0060          ! LOAD IDLE DBR ON CURRENT VP !
00A4 6135  0010'         LD      R3, #IDLE_VP*SIZEOF VP_TABLE
00A8 6F25  0010'         LD      R5, VPT.VP.DBR(R3)
                         LD      VPT.VP.DBR(R2), R5

                         ! TURN ON CURRENT VP'S IDLE FLAG !
00AC 4D25  0016'         LD      VPT.VP.IDLE_FLAG(R2), #ON
00B0 FFFF

                         ! SET VP TO READY STATE !
00B2 4D25  0014'         LD      VPT.VP.STATE(R2), #READY
00B6 0001

                         ! SCHEDULE FIRST ELIGIBLE READY VP !
00B8 5F00  0000'         CALL GETWORK  !(R14: DBR) !

                         ! UNLOCK VPT !
00BC 4D08  0000'         CLR  VPT.LOCK

00C0 9E08                RET
00C2                 END IDLE
               !PAGE
```

```
                    SWAP_VDBR  PROCEDURE
00C2                           !************************!
                               ! LOADS NEW DBR ON       !
                               ! CURRENT VP. CALLED BY  !
                               ! TC_GETWORK.            !
                               !************************!
                               ! REGISTER USE           !
                               !  PARAMETERS            !
                               !   R1: NEW_DBR (INPUT)  !
                               !  GLOBAL VARIABLES      !
                               !   R14: DBR             !
                               !  LOCAL VARIABLES       !
                               !   R2: CURRENT_VP       !
                               !   R4: VPT.LOCK ADDR    !
                               !************************!
                    ENTRY
                       ! LOCK VPT !
00C2 7604  0000'       LDA     R4, VPT.LOCK
00C6 5F00  0150'       CALL    SPIN_LOCK  ! (R4:^VPT.LOCK) !
                       ! NOTE: RETURNS WHEN VPT S LOCKED BY THIS VP. !

                       ! GET CURRENT VP !
00CA 6102  0002'       LD      R2, VPT.RUNNING_LIST

                                       ! * * * DEBUG * * * !
00CE 4D21  001E'                       CP VPT.VP.MSG_LIST(R2), #NIL
00D2 FFFF
00D4 5E06  00E4'                       IF NE ! MSG WAITING !  THEN
00D8 2100  0005                         LD  R0, #SWAP_NOT_ALLOWED
00DC 7601  00DC'                        LDA R1, $  !PC!
00E0 5F00  A900                         CALL MONITOR
                                       FI
                                       ! * * END DEBUG * * !
```

```
                                ! SET DBR !
00E4 612E  0010'                LD      R14, VPT.VP.DBR(R2)

                                ! LOAD NEW DBR ON CURRENT VP !
00E8 6F21  0010'                LD      VPT.VP.DBR(R2), R1

                                ! TURN OFF IDLE FLAG !
00EC 4D25  0016'                LD      VPT.VP.IDLE_FLAG(R2), #OFF
00F0 0000

                                ! SET VP TO READY STATE !
00F2 4D25  0014'                LD      VPT.VP.STATE(R2), #READY
00F6 0001

                                ! SCHEDULE FIRST ELGIBLE READY VP !
00F8 5F00  0000'                CALL GETWORK  ! (R14:DBR) !

                                ! UNLOCK VPT !
00FC 4D08  0000'                CLR  VPT.LOCK

0100 9E08                       RET
0102                        END SWAP_VDBR
                 !PAGE
```

```
!
0102                    TEST_PREEMPT        PROCEDURE
                                            !******************************!
                                            ! TESTS FOR PREEMPT INTERRUPT  !
                                            ! FLAG AND HANDLES INTERRUPT   !
                                            ! IF FLAG IS SET.              !
                                            ! INVOKED UPON EVERY EXIT FROM !
                                            ! KERNEL.                      !
                                            !******************************!
                                            ! REGISTER USE                 !
                                            !  LOCAL VARIABLES             !
                                            !   R1: PREEMPT_INT_FLAG       !
                                            !   R2: CURRENT_VP             !
                                            !******************************!
                        ENTRY
                           TEST_FLAG:
                            DO  ! WHILE CURRENT_VP'S PREEMPT FLAG IS ON !

                    ! NOTE: NEXT TWO STATEMENTS MAY NOT BE RACE FREE.
                           LOCK MAY BE REQUIRED HERE FOR MULTIPROCESSOR SYS. !

                             ! GET CURRENT_VP !
0102 6102  0002'             LD R2, VPT.RUNNING_LIST

                             ! TEST PREEMPT INTERRUPT FLAG !
0106 6121  0018'              LD   R1, VPT.VP.PREEMPT(R2)
010A 0B01  0000              CP R1, #OFF
010E 5E0E  0116'             IF EQ  !  PREEMPT FLAG IS OFF !  THEN
0112 5E08  0122'              EXIT FROM TEST_FLAG
                             FI

                           ! *** VIRTUAL PREEMPT HANDLER *** !
                           ! ** NOTE: SAFE SEQUENCE AND DOES NOT REQUIRE
                                     VPT TO BE LOCKED. ** !
```

```
                        ! RESET PREEMPT FLAG !
0116 4D25  0018'             LD VPT.VP.PREEMPT(R2), #OFF
011A 0000

                             ! SIMULATE PREEMPT INTERRUPT !
011C 5F00  A828              CALL TC_PREEMPT_HANDLER
                  ! ** NOTE: THIS JUMP TO AN UPPER LEVEL (TRAFFIC_CONTROL)
                        IS USED ONLY IN THE CASE OF A PREEMPT INTERRUPT,
                        AND SIMULATES A HARDWARE INTERRUPT. ** !

                          ! *** END VIRTUAL PREEMPT HANDLER *** !

0120 E6F0                 OD

                        ! RETURN TO GATEKEEPER !
0122 9E08               RET

0124                 END TEST_PREEMPT
              !PAGE
```

```
!
0124                    RUNNING_VP    PROCEDURE
                                      !*******************************!
                                      ! CALLED BY TRAFFIC CONTROL.    !
                                      ! RETURNS VP_ID. RESULT IS VALID!
                                      ! ONLY WHILE APT IS LOCKED.     !
                                      !*******************************!
                                      ! REGISTER USE                  !
                                      !  PARAMETERS                   !
                                      !   R1: VP_ID  (RETURNED)       !
                                      !  LOCAL VARIABLES              !
                                      !   RRØ: DIVIDEND               !
                                      !    RØ: REMAINDER              !
                                      !    R1: QUOTIENT               !
                                      !*******************************!
                           ENTRY
                              ! LOCK VPT !
Ø124 76Ø4  ØØØØ'              LDA     R4, VPT.LOCK
Ø128 5FØØ  Ø15Ø'              CALL    SPIN_LOCK  ! (R4:^VPT.LOCK) !
                              ! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP. !

Ø12C 61Ø1  ØØØ2'              LD      R1, VPT.RUNNING_LIST
Ø13Ø BDØØ                     LDK     RØ, #Ø

                              ! CONVERT VP_INDEX TO VP_ID !
Ø132 1BØØ  ØØ2Ø               DIV     RRØ, #SIZEOF VP_TABLE

```

```
                                         ! * * * DEBUG * * * !

0136 0B00  0000                          CP  R0, #0
013A 5E06  014A'                         IF NE  !REMAINDER <> 0 !  THEN
013E 2100  0006                           LD R0, #VP_INDEX_ERROR
0142 7601  0142'                          LDA R1, $
0146 5F00  A900                           CALL MONITOR
                                         FI

                                         ! * * END DEBUG * * !
014A 4D08  0000'          CLR     VPT.LOCK

014E 9E08                 RET
0150                   END RUNNING_VP


                  !PAGE
```

```
0150                  SPIN_LOCK     PROCEDURE
                                    !***********************!
                                    ! USES SPIN_LOCK MECH.  !
                                    ! LOCKS UNLOCKED DATA   !
                                    ! STRUCTURE (POINTED TO !
                                    ! BY INPUT PARAMETER).  !
                                    !***********************!
                                    !REGISTER USE           !
                                    ! PARAMETERS            !
                                    !  R4: LOCK ADDR (INPUT)!
                                    !***********************!
                          ENTRY
                           ! ** NOTE: SINCE ONLY ONE PROCESSOR CURRENTLY IN
                                         SYSTEM, LOCK NOT NECESSARY. ** !
                                              ! * * * DEBUG * * * !
0150 0D41  0000                               CP  @R4, #OFF
0154 5E06  0164'                              IF NE ! NOT UNLOCKED !  THEN
0158 2100  0000                                LD  R0, #UNAUTH_LOCK
015C 7601  015C'                               LDA  R1, $
0160 5F00  A900                                CALL MONITOR
                                              FI
                                              ! * * END DEBUG * * !
                             TEST_LOCK:
                              !  DO WHILE STRUCTURE LOCKED !
0164 0D46                    TSET    @R4
0166 E5FE                    JR MI, TEST_LOCK
                                    ! ** NOTE SEE PLZ/ASM MANUAL
                                             FOR RESTRICTIONS ON
                                             USE OF TSET. ** !
0168 9F08                    RET

016A                      END SPIN_LOCK



                END INNER_TRAFFIC_CONTROL
```

APPENDIX B

```
Z8000ASM  2.02
LOC    OBJ CODE    STMT SOURCE STATEMENT

                      1 TRAFFIC_CONTROL MODULE
                      2
                      3    ! VERS 4 !
                      4
                      5   CONSTANT
                      6   ! ********** SUCCESS CODES ********** !
                      7              ADVANCED            :=0
                      8              EVENT_NOT_FOUND     := 1
                      9
                     10   ! *********** DEBUG CODES *********** !
                     11              BLOCKED_LIST_ERROR := 0
                     12              READY_LIST_ERROR   := 1
                     13              RUNNING_LIST_ERROR := 2
                     14
                     15    ! ******** SYSTEM PARAMETERS ******** !
                     16              NR_PROCESSES       :=4
                     17              NR_MMU_REG         :=64
                     18              NR_VP              :=4
                     19              NR_AVAIL_VP        :=2
                     20              STACK_SEG          :=1
                     21              STACK_SEG_SIZE     :=%100
                     22
                     23        ! * * OFFSETS (FROM TOP OF STACK) * * !
                     24                    PROCESS_ID  := STACK_SEG_SIZE-%1E
```

```
! ******** TEMP PROCEDURE DEFS ******** !
        ! (JUMP_TABLE.4) !
        ITC_SET_PREEMPT := %A808
        ITC_SWAP_VDBR   := %A80C
        ITC_IDLE        := %A810
        ITC_RUNNING_VP  := %A818

  ! ********* SYSTEM CONSTANTS ******** !
        TRUE     := 1
        FALSE    := 0
        ON       := %FFFF
        OFF      := 0
        EVENT_R  := 0
        EVENT_W  := 1
        RUNNING:= 0
        READY    := 1
        BLOCKED:= 2
        IDLE     := %DDDD
        NIL      := %FFFF
        INVALID:= %EEEE
        MONITOR := %A902 ! HBUG ENTRY !

!PAGE
```

```
TYPE

  AP_POINTER    WORD
  ADDRESS       WORD

   EVENT_TABLE  RECORD
        [ HANDLE          WORD
          EVENT           WORD
          TICKET          WORD
          FILLER_2        ARRAY [5 WORD]
        ]
  AP_TABLE RECORD
      [ DBR               ADDRESS
        PRI               INTEGER
        STATE             INTEGER
        NEXT_AP           AP_POINTER
        FILLER_1          ARRAY [4 WORD]
        EVENTCOUNT        EVENT_TABLE
      ]
  MMU_TABLE RECORD
    [   BASE_ADDR         WORD
        ATTRIBUTES        WORD
        FILLER_3          ARRAY [6 WORD]
    ]
  KST_TABLE RECORD
    [ ASTE_NO   WORD
      CLASS     WORD
      FILLER_4  ARRAY [6 WORD]
    ]
```

```
        GAS_TABLE RECORD
            [  GAST_LOCK        WORD
               EVENT_1 WORD
               EVENT_2 WORD
               TCKT    WORD
               FILLER_5  ARRAY [4 WORD]
            ]
        RUNNING_ARRAY ARRAY [NR_AVAIL_VP WORD]

    $SECTION TC_DATA
    INTERNAL

0000    APT RECORD
           [   SUCCESS_CODE     WORD
               LOCK             WORD
               RUNNING_LIST     RUNNING_ARRAY
               READY_LIST       WORD
               BLOCKED_LIST     WORD
               FILLER           ARRAY [2 WORD]
               AP       ARRAY  [NR_PROCESSES AP_TABLE]
            ]
0090    KST ARRAY [NR_MMU_REG MMU_TABLE]

0490    GAST ARRAY [NR_PROCESSES*NR_MMU_REG GAS_TABLE]

    ! PAGE
```

```
!
                $SECTION TC_INT_PROC

0000              GETWORK       PROCEDURE
                                !***********************!
                                ! LOADS NEXT READY DBR  !
                                ! ON CURRENT VP.        !
                                !***********************!
                                ! REGISTER USE          !
                                ! PARAMETERS (INPUT)    !
                                !  R1: CURRENT_VP_ID    !
                                !  LOCAL VARS           !
                                !   R2: NEXT_AP         !
                                !   R3: VP_PTR          !
                                !***********************!
                     ENTRY
0000 6102  0008'        LD      R2, APT.READY_LIST
                        READY_AP_SEARCH:
                         DO  ! WHILE NOT (END LIST OR READY_PROCESS) !
0004 0B02  FFFF           CP    R2, #NIL
0008 5E0E  0010'          IF EQ  ! IF NO READY PROCESSES !  THEN
000C 5E08  0026'           EXIT FROM READY_AP_SEARCH
                          FI

0010 4D21  0014'          CP    APT.AP.STATE(R2), #READY
0014 0001
0016 5E0E  001E'          IF EQ  ! IF PROCESS READY !  THEN
001A 5E08  0026'           EXIT FROM READY_AP_SEARCH
                          FI
```

```
                          ! GET NEXT READY AP !
001E 6123  0016'           LD    R3, APT.AP.NEXT_AP(R2)
0022 A132                  LD    R2, R3
0024 E8EF                 OD

0026 0B02  FFFF          CP      R2,#NIL
002A 5E0E  003C'         IF EQ ! IF NO PROCESSES READY ! THEN
                          ! LOAD IDLE PROCESS !
002E 4D15  0004'          LD     APT.RUNNING_LIST(R1), #IDLE
0032 DDDD
0034 5F00  A810           CALL ITC_IDLE
0038 5E08  004E'         ELSE
                          ! LOAD FIRST READY AP !
003C 6F12  0004'          LD     APT.RUNNING_LIST(R1), R2
0040 4D25  0014'          LD     APT.AP.STATE(R2), #RUNNING
0044 0000
0046 6121  0010'          LD     R1, APT.AP.DBR(R2)
004A 5F00  A80C           CALL ITC_SWAP_VDBR  !(R1:DBR)!
                         FI
004E 9E08                RET
0050                 END GETWORK

                 !PAGE
```

```
!
0050                    TC_PREEMPT_HANDLER  PROCEDURE
                                            !**********************!
                                            ! LOADS FIRST READY AP  !
                                            ! IN RESPONSE TO PREEMPT!
                                            !INTERRUPT              !
                                            !**********************!
                                            !GLOBAL (TC) VARIABLES  !
                                            ! R12: CURRENT_PROCESS  !
                                            ! R13: SEG_BASE_ADDR    !
                                            ! R14: DBR              !
                                            !**********************!
                         ENTRY
                            !** CALL WAIT_LOCK (APT^.LOCK) **!
                            !** RETURNS WHEN PROCESS HAS LOCKED APT **!

                            ! GET RUNNING_VP ID !
0050 5F00  A818             CALL ITC_RUNNING_VP !(RETURNS: R1:VP_ID)!

                            ! GET AP !
0054 6112  0004'            LD      R2, APT.RUNNING_LIST(R1)

                            ! IF NOT AN IDLE PROCESS, SET IT TO READY !
0058 0B02  DDDD             CP      R2, #IDLE
005C 5E06  0066'            IF NE ! NOT IDLE !  THEN
0060 4D25  0014'             LD     APT.AP.STATE(R2), #READY
0064 0001
                            FI

                            ! LOAD FIRST READY PROCESS !
0066 5F00  0000'            CALL GETWORK !(R1: VP_ID)!
```

```
                 !** CALL WAIT_UNLOCK (APT^.LOCK) **!
                 !** RITURNS WHEN PROCESS HAS UNLOCKED APT **!
                 !** AND ADVANCED ON THIS EVENT **!

006A 9E08        RET
006C           END TC_PREEMPT_HANDLER

         END TRAFFIC_CONTROL

   Ø errors
Assembly complete
```

APPENDIX C

```
ADVANCE Procedure (HANDLE, INSTANCE)

 Begin

  Call WAIT_LOCK (APT)

  ! wake up !

  PROCESS := EVENT_LIST_HEAD (HANDLE, INSTANCE)

  COUNT := MM_ADVANCE_COUNT (HANDLE, INSTANCE)

  ! make ready !

  Do while not end of READY_LIST
   If PROCESS.COUNT <= COUNT  THEN
    Call MAKE_READY
   end if
  end do

  ! initialize preempt array !

  Do for VP_ID = 1 TO #NR_VP
   RUNNING_LIST [VP_ID].PREEMPT := #TRUE
  end do

  ! find preempt candidates !

  CANDIDATES := ∅

  PROCESS := READY_LIST_HEAD

  Do  (for VP_ID := 1 to #NR_VP) and not end READY_LIST

   If PROCESS = #RUNNING   THEN
    RUNNING_LIST [VP_ID}.PREEMPT := #FALSE
   else
    CANDIDATE := CANDIDATE +1
   end if

  Get next ready process
 end do
```

```
 ! preempt candidates !

 Do for VP_ID := 1 to CANDIDATES
  If RUNNING_VP [VP_ID] = #TRUE  Then
   Call SET_VPREEMPT (VP_ID)
  end if
 end do

 Call WAIT_UNLOCK (APT)

 Return

End ADVANCE
```

```
AWAIT Procedure (HANDLE, INSTANCE, COUNT)
 Begin
  Call WAIT_LOCK (APT)
  VP_ID := RUNNING_VP
  PROCESS := RUNNING_LIST [VP_ID]
  CURRENT_COUNT := MM_READ_COUNT (HANDLE, INSTANCE)
  If CURRENT_COUNT < COUNT  Then
   Call THREAD_BLOCKED_LIST (HANDLE, INSTANCE, PROCESS)
   PROCESS.HANDLE := HANDLE
   PROCESS.INSTANCE := INSTANCE
   PROCESS.COUNT := COUNT
   PROCESS.STATE := #BLOCKED


   Call TC_GETWORK
  end if
  Return
 End AWAIT
```

LIST OF REFERENCES

1. Coleman, A. A., Security Kernel Design for a Microprocessor-based Multilevel Archival Storage System, MS Thesis, Naval Postgraduate School, December 1979.

2. Parks, E. J., The Design of a Secure File Storage System, MS Thesis, Naval Postgraduate School, December 1979.

3. Moore, E. E. and Gary, A. V., The Design and Implementation of the Memory Manager for a Secure Archival Storage System, MS Thesis, Naval Postgraduate School, June 1980.

4. O'Connell, J. S., and Richardson, L. D., Distributed Secure Design for a Multi-Microprocessor Operating System, MS Thesis, Naval Postgraduae School, June 1979.

5. Schell, LTCOL R. R., Security Kernels: A Methodical Design of System Security, USE Technical Papers (Spring Conference, 1979). pp. 245-250, March 1979.

6. Schiller, W. L., The Design and Specification of a Security Kernel for the PDP-11/45. ESD-TR-75-69, The MITRE Corporation, Bedford, Mass., May 1975.

7. Lampson, B. W., Protection, Proc. Fifth Princeton Symposium on Information Sciences and Systems, Princeton U., March 1971, pp. 437-443.

8. Dijkstra, E. W., "The Structure of the 'THE' Multiprogramming System", Communications of the ACM, v. 11, p. 341-346, May 1968.

9. Madnick, S. F. and Donovon, J. J., Operating Systems, McGraw Hill, 1974.

10. Saltzer, J. H., Traffic Control in a Multiplexed Computer System, Ph.D. Thesis, Massachusetts Institute of Technology, July 1966.

11. Reed, D. P., and Kanoda, R. K., "Synchronization with Eventcounts and Sequencers", Communications of the ACM, V. 22, No. 2, February 1979, p. 115-123.

12. Reed, D. P., Processor Multiplexing in a Layered Operating System, MS, Massachusetts Institute of Technology, MIT/LCS/TR-164, 1976.

13. Jensen, R. W., and Tonies, C. C., Software Enginerring, Prentice-Hall, Inc., 1979.

14. Dijkstra, F. W., "The Humble Programmer", Communications of the ACM, V. 15, No. 10, p. 859-866, October 1972.

15. Schroeder, M. D., Clark, D. D., and Saltzer, J. H., The Multics Kernel Design Project, Paper presented at ACM Symposium, November, 1977.

16. Schroeder, M. D., "A Hardware Architecture for Implementating Protection Rings", Communications of the ACM, V. 15, No. 3, p. 157-170, March 1972.

17. Peuto, B. L., "Architecture of a New Microprocessor", Computer V. 12, No. 2, p. 10-20, February 1979.

18. Organick, E. I., The Multics System: An Examination of its Structure, MIT Press, 1972.

19. Wasson, W.J., Detailed Design of the Kernel of Real-time Multiprocessor Operatiing System, MS Thesis, Naval Postgraduate School, June 1980.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
4.	Prof. Lyle A. Cox, Jr., Code 52C1 Department of Computer Science Naval Postgraduate School Monterey, California 93940	4
5.	LTCOL Roger R. Schell, Code 52Sj Department of Computer Science Naval Postgraduate School Monterey, California 93940	5
6.	Joel Trimble, Code 221 Office of Naval Research 800 North Quincy Arlington, Virginia 22217	1
7.	LT Alan V. Gary 3320 W. Epler Ave. Indianapolis, Indiana 46217	1
8.	LCDR Edmund E. Moore NAVELEXSYSCOM PME 107 Washington, D.C. 20360	1
9.	CAPT John L. Ross 107 Headon St. Weatherford, Texas 76086	1
10.	LT Hal R. Powell 1295 Heatherstone Way Sunnyvale, California 94087	1

11. Office of Research Administriation 1
Code 012A
Naval Postgraduate School
Monterey, California 93940

12. Prof. Uno R. Kodres, Code 52Kr 1
Department of Computer Science
Naval Postgraduate School
Monterey, California 93940

13. I. Larry Avrunin, Code18 1
DTNSRDC
Bethesda, Maryland 20084

14. R. P. Crabb, Code 9134 1
Naval Oceans Systems Center
San Diego, California 92152

15. Kathryn Heninger, Code 7503 1
Naval Research Lab
Washington, D.C. 20375

16. Dr. J. McGraw 1
U.C. - L.L.L. (1-794)
P.O. Box 808
Livermore, California 94550

17 Mark Underwood 1
NPRDC
San Diego, California 92152

18. Walter P. Warner, Code K70 1
NSWC
Dahlgren, Virginia 22448

19. M. George Michael 1
U.C. - L.L.L. (L-76)
P.O. Box 808
Livermore, California 94550

20. LCDR Stephen L. Reitz
NAVSEA TECHREP
St. Paul, Minnesota 30845 2

LMED

8