NPS52-80-002

NAVAL POSTGRADUATE SCHOOL

Monterey, California

The Naval Postgraduate School
SECURE ARCHIVAL STORAGE SYSTEM
Part I - Design -

Roger R. Schell and Lyle A. Cox

March 1980

proved for public release; distribution unlimited

epared for:

Chief of Naval Research
Arlington, Virginia 22217

26 Mar 1980

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral J. J. Ekelund
Superintendent

Jack R. Borsting
Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NPS52-80-002	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): The Naval Postgraduate School SECURE ARCHIVAL STORAGE SYSTEM Part I - Design -	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Roger R. Schell and Lyle A. Cox, Jr.	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School Monterey, CA 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61152N; RR000-01-10 N0001480WR00054
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School Monterey, CA 93940	12. REPORT DATE: March 1980
	13. NUMBER OF PAGES: 325
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Chief of Naval Research Arlington, Virginia 22217	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)

Security Kernel
Archival Storage
Computer Networks
Operating Systems
Computer Security
Microcomputers

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

There is an increasing need for systems which provide controlled access to multiple levels of sensitive data and information. This report comprises the first phase of the realization of such a system: the comprehensive design of a multilevel secure file storage system. This is the focus of an ongoing research project, which is currently in the early implementation phases. The design is based upon security kernel technology as applied to modern multiple microcomputer arrays.

This design is intended to interface with other (distributed) processing

elements, perhaps forming the central hub of a data secure network of computers. The design would provide archival shared storage while insurin that each interfacing processor accessed only that information appropriate The design phase of the project is presented in a series of three research reports (Masters Degree theses). These reports, reprinted in their entire here are: (1) O'Connell and Richardson's definition of a secure multi-microprocessor family of operating systems; (2) Coleman's detailed securi kernel design for a member of this family; and (3) Parks' hierarchical fil system designed to run under the control of Coleman's security kernel.

The Naval Postgraduate School

SECURE ARCHIVAL STORAGE SYSTEM

Part I - Design -

by

Roger R. Schell and Lyle A. Cox

editors

Department of Computer Science

Naval Postgraduate School

Monterey, California

March 1980

ABSTRACT

There is an increasing need for systems which provide controlled access to multiple levels of sensitive data and information. This report comprises the first phase of the realization of such a system: the comprehensive design of a multilevel secure file storage system. This is the focus of an ongoing research project, which is currently in the early implementation phases. The design is based upon security kernel technology as applied to modern multiple microcomputer arrays.

This design is intended to interface with other (distributed) processing elements, perhaps forming the central hub of a data secure network of computers. The design would provide archival shared storage

while insuring that each interfacing processor accessed only that information appropriate. The design phase of the project is presented in a series of three research reports (Masters Degree theses). These reports, reprinted in their entirety here are: (1) Capt. O'Connell and Lt. Richardson's definition of a secure multi-microprocessor family of operating systems; (2) Capt. Coleman's detailed security kernel design for a member of this family; and (3) Lt. Parks' hierarchical file system designed to run under the control of Capt. Coleman's security kernel.

BACKGROUND

The Secure Archival Storage System (SASS) system was first conceived as a research project at the Naval Postgraduate School in late 1978. Growing out of the requirements for unifying multiple computer systems' data bases in a secure environment, for example the diverse resources of the Naval Postgraduate School and its Computer Center, its Computer Laboratory, and its Microcomputer Facility, a research program was initiated to apply existing techniques to state of the art microcomputer architectures. Requirements and experience were drawn from a number of programs, including the security kernel design of Multics [1], the network storage of several systems including OCTOPUS [2], the file structure of several timesharing systems including UNIX [3], and the current operational Navy programs of SNAP2 (the shipborne administrative processing system) and the Navy Laboratory Computer Network (NALCON).

The requirements definition and design phase of this research are just being completed. Prototype implementation of a demonstration

system is beginning, and will be the subject of a second report.

The SASS is designed as a member of the family of secure, multi-microcomputer operating systems described by Capt. O'Connell and Lt. Richardson (Appendix A). It is a rather restricted subset with a uniprocessor, a static set of processes, non-demand memory management, and no application programs besides the supervisor itself. However, a strong family tie is maintained since this project is just one part of research in secure operating systems for multi-microcomputers.

The SASS design effort focuses on what are perceived to be the key research issues and therefore does not include all the features that might be desired for a production version. The design has made a conscious effort to be extendable, and some of these capabilities will be addressed in the future.

A security kernel [4] is used to provide an extended virtual machine supporting asynchronous processes and a segmented virtual memory. The security kernel is responsible for non-discretionary security. It must be capable of enforcing of the DoD classification and clearance policy, but is applicable to other policies such as privacy. Particular attention has been given to minimizing the size and complexity of the kernel. Each host is connected to the SASS with its own bidirectional digital link for the transmission of commands and files. The SASS provides each host with a hierarchical file system. The host can store and retreive files, and can share files with other hosts. The

SASS is self sufficient in the sense that any host can effectively use it, even if the host has no independent knowledge of its contents.

OVERVIEW OF STRUCTURE AND ORGANIZATION

The following sections summarize the salient features and structure of NPS-SASS. Design details for each area are found in Appendices A thru C.

STORAGE SYSTEM STRUCTURE

To minimize the security kernel, the storage system structure is created entirely in the supervisor domain that is "outside of" the kernel, as described by Lt. Parks (Appendix C). The file system supervisor builds on the primitive process and segment objects created by the kernel.

Internal Organization

The file system supervisor creates a single, tree structured file system using only the primitive segments provided by the kernel. Some segments are used to create directories whose contents are managed by the file system itself. These directories define the hierarchical file structure. The underlying kernel has no explicit knowledge of this file structure.

The files from a host are stored as a strings of bits that are never interpreted in any way within the SASS. One or more segments, as required, is used to store each file.

Two processes are associated with each host link and access that portion of the file structure associated with that host. An I/O process provides for the transmission of information over the link to the host. This process communicates with a file manager process that is responsible for the file system structure.

Host Interface

The I/O process communicates with its host using fixed length packets. The physical I/O operations are performed by the kernel, in response to requests from the supervisor. Packet exchange with the host is on an asynchronous (send/acknowledge) basis with a limited "send ahead", error checking, and retransmission if required.

There are two classes of packets: data and commands. Each command to the SASS is an "atomic" operation and addresses a single file. The commands are actually executed by the file manager process. The design specifically addresses the problem of retaining a valid file, even if the SASS is interrupted (e.g. a power failure) in the middle of a command.

Host Storage

A SASS file manager process creates a virtual file system for each host link. The host accesses files by means of commands, e.g., to create, delete, store or retrieve files. Each host virtual file system is essentially a subtree of the total SASS hierarchy.

In addition, the virtual file system can include a file link to a file of another host. This is used for sharing files. The SASS provides conflict free sharing, even if a shared file is retrieved by one host while it is being updated by another host. The retrieved file will be some complete and valid version of the file that existed between the time of the retrieval command and the time of the command completion acknowledgement by the SASS.

SECURITY KERNEL ORGANIZATION

All the physical hardware resources are managed by the kernel of the operating system, as described by Capt. Coleman (Appendix B). The initial design for the SASS uses a serial, RS232 link to each host. It contemplates a hard (Winchester) disk as the actual storage medium, although care has been taken to avoid device dependence outside the device driver itself. The kernel transforms the physical resources into virtual resources for the use of each supervisor process.

Segmented Memory

The kernel produces a segmented (virtual) memory. This is done through the use of memory management hardware, such as the memory management unit of the Zilog Z8000 [5] or the contemplated Intel VLSI follow on [6] to the 8086.

This is a non-random access virtual memory. There is no demand memory management. A supervisor process must request that the kernel swap specific segments into or out of memory. Each process is allocated

a bounded, linear quantity of virtual memory into which it can swap segments. All communication between processes is accomplish using shared segments.

Asynchroncus Processes

As noted before, there are two supervisor processes for each host link. The kernel includes synchronization primitives to facilitate communication between the I/O and file manager processes of a host. In addition the file manager processes of different hosts use these synchronization primitives to control race condition when accessing shared files. The I/O process also uses this to synchronize with the physical transmission of packets over the host link.

There is also a memory management kernel process. This process does the actual I/O needed for the swapping of segments between secondary storage and memory.

SECURITY

The most distinctive feature of the SASS is its methodical treatment of security. The security kernel technology is applied, and this has strongly influenced both hardware and software decisions. The security kernel is organized as a distributed operating system for the supervisor processes; that is, its functions are distributed in all processes.

Each process has two domains: supervisor and kernel.

Protection of the Kernel

Since it is included in the address space of every process the security kernel procedures and data are to be protected from tampering by the supervisor. This means there must be some form of hardware enforced domains. Since there is a strictly hierarchical relationship (kernel more privileged than supervisor), protection rings (as defined for Multics) are a satisfactory domain implementation. In fact, for only two domains, simple user and supervisor processor modes can be used along with memory management hardware to realize two rings.

The design includes a software gatekeeper to manage the entry of a process into the kernel, viz., when the supervisor calls on a kernel function - this gatekeeper also provides parameter validation for these cross-ring calls.

Non-Discretionary Security

A security policy may require enforcement of access limitations that are established external to any computer. The DoD classification and clearance policy is a common example of such a non-discretionary policy. The security kernel is responsible for this enforcement. In fact, a major design goal has been to simplify the kernel as much as possible by including only those functions necessary to enforce non-discretionary security.

The link to each host is for a single access class (e.g., single

level of DoD classification). This access class is authenticated by a physical connection, that may include an encrypted communication path. If a host is multilevel secure, then it may have multiple host links to the SASS. Each SASS process, of course, is assigned the access class of its corresponding host link. An example of the impact of security is that process synchronization must be done with eventcounts [7] rather than more traditional mechanisms such as P and V.

Discretionary Security

In addition to the non-discretionary limitation, file access can be further controlled based on individual user identification. This discretionary security is enforced outside the kernel by the file management processes. Each file has an access control list specifying its authorized users.

The user is identified (by the host) as part of each command to the SASS. Clearly the reliability of this control is limited by the ability of a host to reliably authenticate users and pass their identity to the SASS. However, no host weakness can impact the reliable enforcement of the non-discretionary controls by the kernel.

Verifiability

The research goals of this project do not include verification methodology that is being addressed by several other groups. Use of the SASS in a hostile environment would most likely be preceeded by a formal verification effort that for kernel. The security kernel is designed to

the formal non-discretionary security model, and thus is considered "veritiable." By this we do not mean that the verification would immediately succeed. Rather, we are confident that the problems discovered by verification would not require any basic changes to the design.

SUMMARY

We have attempted to summarize the salient features of the Naval Postgraduate School Secure Archival Storage System Design. We have found this project to be an interesting and exciting experience in applying state of the art security and operating system techniques to the architecture of modern microcomputers. We hope the reader has been encouraged to examine the design details as presented in the following appendices. Finally we would solicit any comments or suggestions the readers might have for our consideration as we continue with implementation.

ACKNOWLEDGEMENTS

The editors would like to acknowledge the many long hours of dedicated effort on the parts of Capt. J. O'Connell, Lt. L. Richardson, A. Capt. Coleman, and Lt. E. Parks which was certainly "above and beyond the call" of their academic requirements. Additionally, we would like to thank Professor Uno Kodres of the Naval Postgraduate School for his support, advice and interest which in many ways enabled us to begin and

pursue this research.

This research was partially supported by grants from the Office of Naval Research Project Number NR 332005 monitored by Mr. Joel Trimble, and from the Naval Postgraduate School Research Foundation.

REFERENCES

[1] Schroeder, M. D. et al, "The Multics Kernel Design Project," Proc. Sixth ACM Symposium on Operating Systems Principles, November 1977, pp 43-56.

[2] Schneider, Thompson and Whitten, "Users Guide to the OCTOPUS Computer Network" University of California Report UCID-30048-R3, October 1976.

[3] Ritchie, D. M., and Thompson, K., "The UNIX Time-Sharing System," Comm. ACM 17,7 (July 1974), 365-375.

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

[5] Peuto, B. L., "Architecture of a New Microprocessor," Computer, 12, 2 (February 1979), 10-21.

[6] Markowitz, R., and Pohlman, W. B., "The Evolution Path of the 8086 Microprocessor Architecture for Operating System Environments,"

Intel Corporation, 1980.

[7] Reed, D. P., and Kanodia, R. K., "Synchronization with Eventcounts and Sequencers," Comm. ACM 22, 2 (February 1979), 115-123.

DISTRIBUTED, SECURE DESIGN FOR A MULTI-MICROPROCESSOR OPERATING SYSTEM

by

James Steven O'Connell
Captain, United States Marine Corps
B.S., University of Utah, 1971

Larry Don Richardson
Lieutenant, United States Navy
B.S., University of Nebraska, 1973

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1979

Dean of Information and Policy Sciences

ABSTRACT

This thesis applies the state of the art techniques for methodical design of secure operating systems to a distributed, multi-microprocessor environment. Explicit process structure and utilization of virtual environments are the fundamental concepts that form a basis for the design presented. The primary design techniques utilized in the design are segmentation, distributed operating system, security kernel, multiprocessing, "cache" memory strategy and multiprogramming. The resulting design is for a family of distributed operating systems that can provide the power of yesterdays large computer in a microprocessor environment. Security, configuration independence, and a loop free structure are the primary characteristics of the design. The design, although hardware independent, was formulated with the Zilog Z8000 or similar microprocessor in mind.

TABLE OF CONTENTS

I. INTRODUCTION

The microprocessors available today are affordable and powerful computing devices. Applying these resources to various applications, especially those requiring multiple microprocessors, presents a formidable problem. The solution to this problem is a family of operating systems to effectively orchestrate processor and memory management across a wide range of applications. However, such systems have not come from the specialized microprocessor operating systems in use today. Such an operating system family could provide a major reduction of overall system software cost in the microprocessor environment.

In this thesis the substantial body of operating system design principles are applied to a methodical design of an operating system for the microprocessor environment. For realism the Zilog Z8000 microprocessor[1] is considered representative of modern features. Configuration independence, distributed processing, multiple protection domains, multiprocessing and multiprogramming are addressed in the design of a secure operating system suitable for a family of operating systems: ranging from a specialized tactical system to a multi-user time sharing system.

The thesis will also identify meaningful subsets of the design (viz., smaller members of the family) for potential use, and state hardware needed (future development) to

implement the design to its fullest capabilities. The operating system designed in this thesis will be referred to as the SYSTEM throughout the thesis.

A. MOTIVATION

The processing power of microprocessors is increasing. If this power could be effectively coordinated by an operating system it could provide a more affordable and powerful product. In addition, there is a growing emphasis on the protection of information stored and processed in computers; hence, the requirement for a system that also provides information security.

The multi-microprocessor systems in use today suffer performance degradation as more processors (generally a maximum of 4 to 5) are added to the system. Sophisticated crossbar interconnections between processors and memories can reduce this problem. However, there is still a need for a combination of microprocessors and memory that do not suffer massive degradation as more processors are added.

The ability to configure a system to meet a variety of capacity needs is an important feature; however as software becomes an increasing portion of system cost, the ability to reconfigure the system as requirements change without major re-design effort is often an even more valuable feature. For this reason the design technique of resource virtualization will be applied as a way to realize configuration independence.

B. BASIC ELEMENTS OF DESIGN

The SYSTEM is composed of a supervisor and a security kernel[2]. The supervisor supports user services (dynamic linking, discretionary security, demand memory management and a hierarchical file system). The security kernel controls the physical system resources (processors, memory, and external devices) to provide virtual resources for the supervisor.

1. Process Structure

A process within the computer system is an internal representation of the computational task of a user utilizing the system. Each process is characterized by an execution point and an address space. Attributes of each process include a security class authorization and a unique identifier that corresponds to the user. By supporting distinct, explicit processes the operating system allows an application to be divided into several cooperating parts. Such a process structure leads to simpler more effective software.

2. Segmented Virtual Memory

Segmentation involves separating all stored information into discrete packages called segments. Each segment has attributes such as security class and access (read or write) permissions. A process' address space is a collection of segments. Segmentation is used by the

supervisor to present the user a random access virtual memory. Copies of all segments are kept on secondary storage until actually referenced, at which time room is made for it in main memory, possibly by removing another segment from memory. This demand memory management is done within the supervisor. The supervisor views a non-random access virtual memory. By presenting the supervisor and the user with virtual environments the kernel establishes configuration independence for them.

3. Distributed Operating System

The address space of each process has three domains (user, supervisor and kernel). The domains form sub-sets of the address space by limiting the segments that can be accessed when the process' execution point is within a given domain. The operating system is part of each process. It is distributed throughout all the processes in protected domains (supervisor domain and kernel domain). Maximum access is in the kernel domain. It is the most priviledged, and the traditional 'privileged instruction' can be executed only in the kernel domain. Only the kernel domain has access to system wide data bases.

The kernel domain creates an extended machine for the supervisor and is supported by system processes. The supervisor is less priviledged but provides the user domain with certain common services such as discretionary security and virtual memory. It should be noted that by distributing

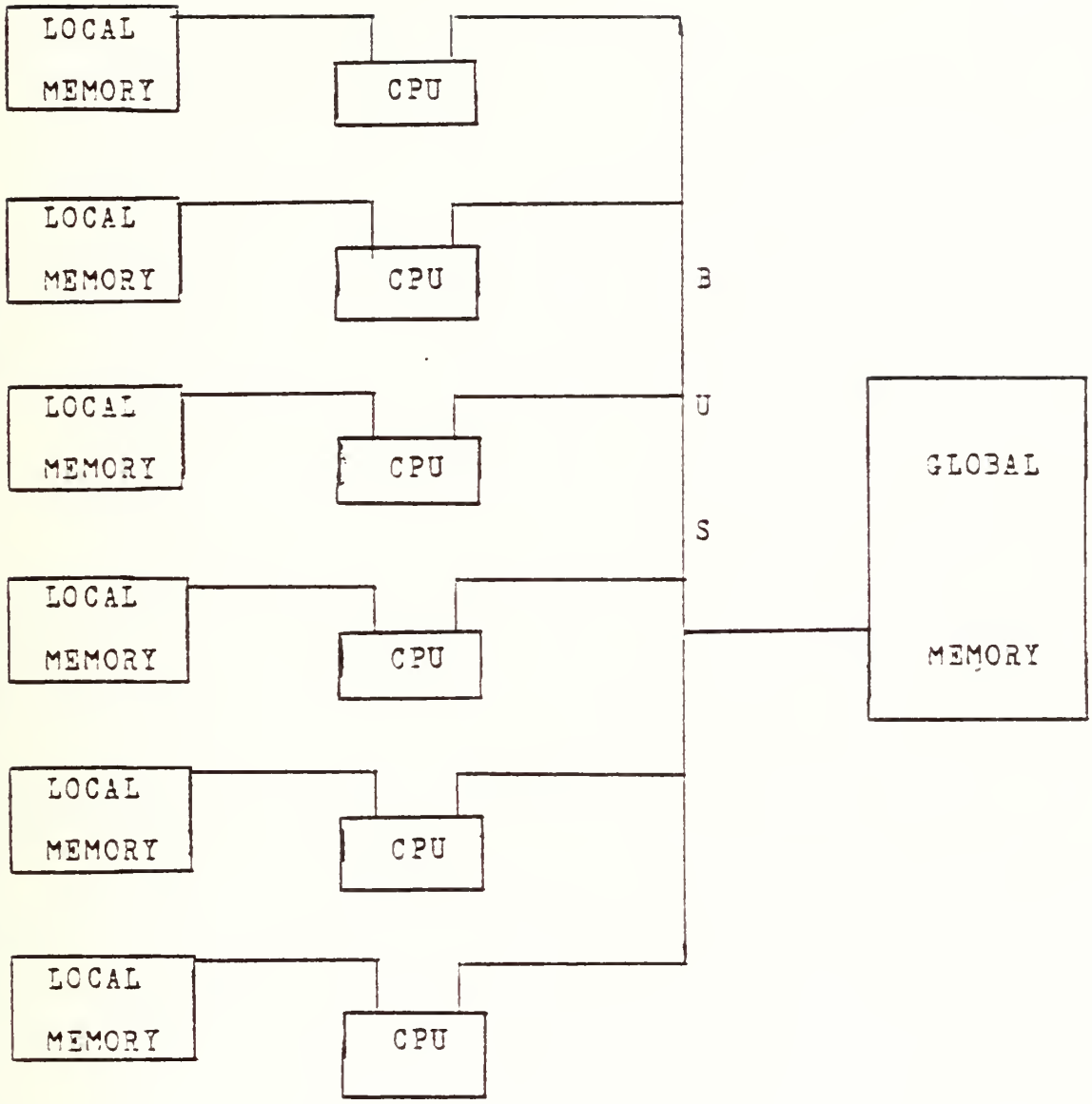
the operating system throughout all processes, services are independently (and simultaneously) available to each process.

4. Processor-Local Memory

The operating system is designed to support a multi-processor configuration with a local memory in close proximity to each processor. The local memory is addressable only by that processor. In addition there is a global memory that is addressable by all processors (Figure 1). Segmentation is the key to effective allocation of information between local and global memory. Problems can arise in the use of a local memory. If a process is allowed to execute on any processor then each time the process is switched from one processor to another the contents of local memory must also be switched. Thus the use of local memory implies that general multiprogramming should not be allowed. This problem can be alleviated by allowing multiprogrammed processes to be semi-dedicated, that is make an effort to restrict the process to a certain processor.

5. Security Kernel

Security cannot in general be built around a present system (i.e., added to) but rather a system must be built around security. Yet today there are a limited number of "secure" systems. One of the main obstacles in providing security is verifying the system is secure. The recently



LOCAL/GLOBAL MEMORY

MULTIPLE CPUs

FIGURE 1

developed security kernel [2] technology has made it possible to solve this problem. By keeping all the things that provide the security in the security kernel and keeping the things that do not involve security out, the security kernel can be kept relatively small and verifiable. The desire to keep the security kernel small (to simplify the verification procedure) is one of the goals driving several design choices.

C. STRUCTURE OF THE THESIS

First, the fundamental concepts (process structure, virtual memory and security) and their relationships to the SYSTEM are discussed. Second, the design of the SYSTEM is presented. This includes a discussion of the design techniques utilized as well as an explanation of the proposed design. Third, the conclusions are presented.

II. FUNDAMENTAL CONCEPTS

A. PROCESS STRUCTURE

By dividing a job into asynchronous parts and executing these parts as seperate entities significant benefits can be realized. Within a single processor system, the partitioning into asynchronous parts provides "only" design simplicity (and thus software economy). In a multi-processor system the partitioning into asynchronous parts is essential if the parallel processing potential of the system is to be realized.

1. Definition of a Process

A process is characterized by an execution point and an address space. Saltzer[3] defines a process as a program in execution on a pseudo-processor. Each process is assigned a unique identifier and is an explicit entity that requires management. In a distributed operating system, those portions of the operating system that are logically part of the sequential flow of control (viz., locus of execution) are within the address space of the user process. This is made possible by dividing the operating system into procedures which are called like any other procedure. It should be noted that in a distributed operating system there is no "master" assigning processes to processors. Rather, each running process "hands off" its processor to the next

process that is to run.

2. Multiple Domains

To protect these procedures from the user, the process' address space is divided into hierarchical domains: user, supervisor, and kernel. The kernel domain is the most privileged. Only the security kernel executes in this domain and can access all segments within the address space. All system wide data bases are restricted to access by the security kernel to prevent any exchange of information between processes, in violation of confinement[4]. There could be more than three domains, and all domains need not be hierarchical, but three is minimum for this design.

The supervisor domain is less priviledged and excludes segments representating the management of the shared resourses. The supervisor domain is separated from the user to protect the user from inadverently destroying the operating system services. The user domain is the least priviledged. The data bases utilized by the supervisor contain only "process local" information - that is, information that is required by this process alone.

Proper controls and checks are utilized when switching the domains (flow of control) so that the security policies are not violated. The hierarchy could be implemented with rings[5] in hardware. Since hardware rings are not available in microprocessors, separate segment descriptors for each domain can be used, with software ring

changes as was done in the original Multics design[6]. The Zilog Z8000 can use multiple memory management units (MMU) to provide the separate descriptor for each domain.

Operating system procedures generally are permitted to reside within the local memory (possibly ROM) of each processor. In the cases of the security kernel, some of the data bases of these procedures are shared by all processors and therefore will reside in global memory. To prevent undesired intervention by simultaneous accesses to these data bases a locking scheme must, of course, be provided. Choosing to put the operating system procedures in each local memory will "waste" memory but may well provide a higher performance by keeping most memory references to local memory where there is no contention for the BUS to global memory. In a specific instance the choice will be determined by whether or not the cost of memory is significant when compared to the value of the increase in performance.

3. Communication and Synchronization

For parallel processing, a job that is composed of a mixture of sequential and non-sequential tasks is explicitly divided into an appropriate structure of processes that can run concurrently. Inter-process communication and synchronization are necessary for parallel processing. Inter-process communication provides synchronization to coordinate the exchange of data between processes. The

actual exchange is realized by use of a shared writable segment. This segment acts like a mailbox in that messages (data) can be delivered by any process that has the appropriate access (both discretionary and non-discretionary).

The synchronization between processes is supported by the BLOCK and WAKEUP, which are kernel calls to the traffic controller. It should be noted that the P and V semaphores[7] are useable for synchronization but were not chosen. The traffic controller concept is taken from Saltzer[3], and his block and wakeup have demonstrated their usefulness in his design for Multics. The traffic controller is the operating system (kernel) module that manages processes. The traffic controller has the job of scheduling user processes. The traffic controller does this by multiplexing the users processes onto a limited number of virtual processors.

The BLOCK and WAKEUP are primitives of the traffic controller that provide synchronization for the user processes. How the user's procedures invoke the BLOCK and WAKEUP primitives will, of course, determine the actual process structure. These primitives can be used to provide simple cooperation, such as mutual exclusion, or complex interactions when required by the application. A process can only block itself and cannot block another process. The block invokes the traffic controller and the traffic controller puts that process in the blocked state and then

schedules another process to run on that virtual processor. The process that is scheduled next is based on the specific scheduling policy of the traffic controller.

The wakeup is used to provide asynchronous processes a synchronization signal. The parameter passed with the wakeup is the process ID of the process for which the wakeup is intended. The wakeup invokes the traffic controller. The traffic controller checks the state of the process specified by the parameter. If that process is not in the blocked state the traffic controller returns, otherwise he will put that process in the ready state and determine if there is another process running with a lower priority. If this is the case the traffic controller will send the virtual processor that the lower priority process is running on a pre-empt interupt, and then return. The process that receives the pre-empt interupt will transfer control to the traffic controller who will in turn schedule the ready process with the highest priority.

Another SYSTEM module concerned with synchronization is the inner traffic controller. This manages the hardware (real) processors to create the virtual processors that are managed by the traffic controller. The inner traffic controller provides the interface between the virtual and physical (real) processors. The inner traffic controller is responsible for assigning the small, fixed number of virtual processors to physical processors. Each physical processor has associated with it several virtual processors. Some

virtual processors are multiplexed between users processes by the traffic controller. The remaining virtual processors are allocated to the system processes. Each system process is assigned a virtual processor. The inner traffic controller determines which virtual processor will run on the physical processor based on the priority assigned to each virtual processor. The primitives SIGNAL and WAIT are used by the inner traffic controller to provide communication and synchronization between the virtual processors. SIGNAL and WAIT are very similar in form and function to BLOCK and WAKEUP, except for the fact that they relate to virtual processors rather than user processes.

4. System Processes

System processes are used to perform operating system functions that are asynchronous to each user process. System processes are typically responsible for the shared resources. The system processes are in the kernel and therefore permitted to access information of any access class. The system processes include the I/O_MANAGER and MEMORY_MANAGER.

5. Process Switching

Process switching is the removing and assigning of processes to virtual processors. When a process switch occurs the execution point (internal registers) and address space of the process being removed must be saved (unloaded),

and then the execution point and address space of the new process must be loaded.

Some systems utilize a descriptor base register (DBR)[6,p.12], which is a pointer to multiple descriptor lists in memory - one list for each process. To change the address space you only need to switch the DBR in the physical processor. However, in microprocessor systems a descriptor list is implemented as registers in the memory management unit (MMU). Process switching can be costly when MMU registers are saved and restored for each change in address space. Alternatively, it is possible to increase the number of MMUs and then the address space could be changed by just switching control to another MMU.

B. SEGMENTED VIRTUAL MEMORY

In many memory handling schemes a user process cannot run until there is sufficient memory available to load its entire address space. This requires large main memory and restricts the size of the process's address space. An alternative is to use the operating system to produce the illusion of an extremely large memory. Since the large memory is merely an illusion, it is called virtual memory. Demand segmentation is a memory management scheme which is used to realize the concept of virtual memory in this design.

Memory has three different views which corresponds to the three different domains (user, supervisor, kernel)

within the computer system. Starting with the user, each view is derived from the previous view by means of an "extended machine" view. The user sees a practically unlimited segmented virtual memory. The user is no longer involved in memory management. Demand memory management is utilized to interface between the user view and the supervisor view.

The supervisor views a fixed amount of virtual memory. The memory is fixed by the physical memory allocated to each process by the kernel. The kernel establishes a mapping between the supervisor memory and the kernel memory. The memory is virtual because there are only absolute addresses in the kernel. The supervisor multiplexes the user's segments onto this fixed virtual memory in response to a hardware fault when the process references a segment that is not in memory. The demand memory management was placed in the supervisor because it is not involved with security and we want to keep the security kernel as simple and small as possible.

The kernel views a fixed physical memory. The physical memory is limited by the local memory available to the processor for use by the user processes. There is some minimum amount of memory required by the operating system for each processor. Before a process is elgible to run, its fixed virtual memory (of the supervisor) must be mapped into the fixed physical memory of the processor. We then call the process "loaded". The kernel's memory manager is responsible

for the proper mapping as the processes address spaces are multiplexed onto the processor's physical memory.

The idea is to require that a limited amount of the job be resident in memory. When the user requests a portion of the process that is not currently in memory, a fault will occur. The supervisor, using the demand memory manager, must find the requested segment and decide where it wishes to place the requested information in virtual memory. The supervisor then sends a request to the kernel to bring this information into memory, thereby repairing the fault so that normal processing can resume.

1. Segmentation

In most micro systems, the user cannot effectively share memory because the different uses of memory can not be specified. The inability to specify the memory use makes memory management difficult, especially when there is memory local to each processor. The different uses are denoted by shared/unshared and writeable/non-writeable (read). The following matrix lists the uses and where they may reside.

	writeable	non-writeable
shared	global	local/global
unshared	local	local

If the memory can be divided by uses and each part has attributes which distinguish the uses, then the management of memory is made reasonable.

Segmentation provides the ability to divide the memory into parts (segments). A segment is a collection of information important enough to be given a name. Each segment is distinguished from others by its logical attributes, that provide the basis for the desired control. Segmentation provides a mechanism for a limited portion of a processes' information to reside in memory at any one time. This also facilitates easy movement of information by segment in and out of memory. The collection of all segments that a process may access (whether or not in physical memory) is what composes its address space.

2. Loading

The loading of a segment consists of finding a segment and making it known (discussed later) to the requesting process (viz., adding the segment to the address space). It is the added feature of segmentation that this loading may be delayed until the segment is actually needed. At that time a segment name can be transformed into a file system pathname. The pathname can then be resolved into the unique identifier for a segment. Then the supervisor requests a segment number be assigned by the kernel, which makes the segment known to the process. If the segment is then required for execution it is physically loaded into memory when actually referenced.

Each segment has associated with it a segment descriptor[6] which contains its attributes (address in

memory, size, access allowed). Since this descriptor is referenced by the hardware at each access request to this segment, then the memory uses can be distinguished. The different segment descriptors of a process can then be contained in a descriptor list. This design utilizes the MMU (memory management unit) which consists of a set of registers to implement the descriptor list. Each register (segment descriptor) contains the descriptor of a particular segment. The MMU registers retain the distinct attributes of each segment at execution time and, therefore, makes it possible for another process to share selected segments, if desired.

The dynamics of a segment fall into two classes, physical and logical. An example of the physical dynamics is the request of a user for write access to a currently used segment. The operating system can physically move the segment from local to global memory so the segment can be shared without the user's knowledge. A stack segment whose size varies is an example of logical dynamics.

3. Dynamic Linking

When a procedure segment makes an external reference to another segment, the address of the later segment must be determined. This is called linking, the constructing of executable instructions that achieve references to external objects (segments). Linking need not be completed at load time. It can be postponed until the actual reference is

encountered. This waiting to link, until referenced, is called dynamic linking[8]. Segmentation is not necessary to achieve dynamic linking, but it helps. When a process begins execution, it should not have to find and bring into memory any more of its segments than is absolutely necessary to begin running. The mere presence of a reference to an external segment in a segments text is no guarantee that the flow of control will touch this reference. Therefore, there is little point to undertake the expense of finding a segment and making it known unless there is some significant expectation that that segment will be referenced during the time allotted to that process. Dynamic linking permits unnecessary linking to be eliminated.

Once the segment has been made known to the process (assigned a segment number), even though it may be moved in and out of memory, the references to this segment need not be changed since the segment number remains the same. The segment descriptor is used to reflect the presence of a segment in memory and the current address in memory. The segment looses none of its attributes by virtue of having been made known to this process.

4. Information Sharing

Segmentation allows direct addressability by the process to any segment within the process' address space. The basic advantage of direct addressability is that the copying of data is no longer mandatory. A segment is also a

unit of sharing. This eliminates the need to duplicate a segment for each requesting process and saves memory. Even more important is the idea that sharing provides a means of inter-process communication. This is important for realizing the power of the explicit process structure, that is essential to an effective multi-processor environment.

In general each procedure segment must be pure to ensure sharing is implemented correctly. A pure procedure operates on variables in registers or in separate data segments associated with the process. It never stores data internally, nor does it alter itself. The linkage segment is such a data segment used to support the pure procedure. A linkage segment is associated with each process. The linkage segment is composed of linkage sections. There is one linkage section for each procedure segment. The linkage section is used to place all alterable information (linkage faults, segment numbers, other static temporary variables) for the pure procedures. Thus, the processes' segments which are pure may be shared while linkage sections must be unique to each process. The fact that the linkage segments are not shared makes it possible to assign different segment numbers to the same procedure in different processes since segment numbers occur explicitly only in linkage segments, that may be different for each process.

The approach in this design is to place the copies of requested segments into local memory, thereby reducing the data bus traffic. If the read-write access requirements

are such that a segment must be physically shared, then it is placed in global memory and every process that is given access will access it there. The key to this memory management is segmentation that keeps a segment's attributes explicit. The kernel can properly manage placement in local and global memory with no intervention from the supervisor or the user to "declare" that the sharing is needed.

5. Access Control

The access control in this design is separated into discretionary (supervisor) and non-discretionary (security kernel). When a segment is requested the supervisor references the access control list attribute for that segment and the access authorized for that process (subject) is determined. The supervisor then passes this to the security kernel so that a non-discretionary check can be made. The kernel compares the access class of the segment with that of the process and the appropriate access is allowed. This access authorized is always the lesser of that requested by the supervisor and that permitted by the kernel. The access one process has for a segment is independent of the access another process has for that same segment.

6. Functional Subsets

Some members of the family of operating systems will not include all of the functions made available by this

design. As an example, consider a family member (e.g. for tactical system) supporting applications that are entirely resident in memory and pre-linked. It would require none of the virtual memory functions provided by the supervisor. This design readily allows this sort of functional subsetting because of its loop free structure[9].

C. SECURITY

The increased capability of the computer system in the last decade has dramatically increased its possible uses. Many users have actively allowed the computer system to assume an increasing number of jobs upon which the user depends to successfully function. As more dependence was placed on the computer it became evident (regrettably by example) that a knowledgeable user (employee of a user) who has access to the computer also has access to all the information contained within the system. Users such as the government (classified information), banking facilities (transfer of funds), corporations (trade secrets) have a need to protect certain information from specific users; therefore, there is an increasing demand for a secure computer system. Designating a specific computer to only run at a specific security class or only running certain security classes at specific times has proven unsatisfactory for the user who has information at many access classes. What is commonly called a "multilevel" environment is one in which information and users at different security classes

can exist simultaneously on the same computer system without permitting a user to access information he is not authorized to use. One goal is to design a system which will allow secure operation in a multilevel environment.

1. Computer Security Problems

The initial attempts to provide a secure system involved adding security onto existing systems. This proved largely useless for designers were intuitively trying to block methods of would-be-penetrators rather than providing a technically sound system design. These futile attempts[10] led to the emerging technique of methodically designing a secure system based on a security kernel derived from a mathematical model (discussed later).

Information security can be provided by external and/or internal control. External control includes guards, watch dogs, door ciphers or anything which would prevent an unauthorized penetration of the compound. Once the penetration is made, the pot of gold is exposed. The internal control is concerned with preventing unauthorized penetration of the computer system. This involves insuring the effectiveness of internal mechanisms in the operating system so that only authorized exchanges of information in a multilevel environment can occur. This includes providing no information to unauthorized users and consistent replies to security violations. The latter is necessary to insure no inadvertant leakage of information[4] concerning the

internal mechanisms. External control is expensive and human-prone. It does not provide for the secure sharing of information needed by many applications, thus forcing users to forego many of the capabilities of modern computers. A goal of this thesis is to design an operating system that provides information security by utilizing internal control. External controls are, of course, still required to physically protect the computer system's information.

The reference monitor is an abstraction created to present the conceptual idea of providing a secure computer system. The reference monitor is composed of subjects, objects, and an access matrix. Subjects are system entities such as a user or a process that can access system resources. Objects are system entities such as data, programs and peripheral devices that can be accessed by subjects. The access matrix represents the permitted accesses between subjects and objects. The reference monitor must support the ability of subjects to reference objects as per the access matrix and it must also support the ability to alter the access matrix.

The security kernel[2] is a relatively recent technical breakthrough for computer security. The security kernel is that portion of the computer's hardware and software which enforces the authorized access relationships between subjects and objects. It is the realization of the abstract concept of a reference monitor. The software portion of the kernel acts as an interface between the rest

of the system and the hardware. The software content of the security kernel is influenced by the hardware features of the processor. The underlying idea is that if the hardware is proven correct and if the software is kept small and it can be proven correct, then we can provide internal security controls that are effective against all possible internal attacks. Global variables such as the unique identifier have been excluded from the supervisor. This has been done to prevent undesired leakage of information. The global variables are placed in the kernel where their proper use can be verified[11].

The security kernel must meet three essential design requirements. First, the kernel must be tamperproof. Second, the kernel must be invoked on every attempt to access information. Every reference must be checked by either software or hardware that is provided with sufficient information to make correct decisions on granting or denying access. Finally, the kernel must be subject to certification. "Subject to certification" implies that the kernel's correctness must be proveable in a rigorous manner using a mathematical model as the basis for the criteria to be met.

In developing a secure system the approach to be followed should consist of the following: determine the security policy to be enforced, develop a mathematical model consistent with desired security policy, design a security kernel based on the mathematical model, implement the design

using available hardware and required software. A computer system is said to be "secure" with respect to some specific security policy. A security policy consists of the external laws, rules and regulations that establish what access is to be permitted. There are two distinct types of security policy: non-discretionary and discretionary.

NON-DISCRETIONARY POLICY involves checking the requested (viz.,the object's) access class (oac) with the access class of the (subject) requestor (sac) to insure they are compatible. Each system contains a lattice structure[12] that defines the relationships between different access classes. The following defines the access permitted:

sac=oac, read/write permitted

sac>oac, read permitted

sac<oac, no access

The lattice can be totally ordered (all classes related) or it can be partially ordered (not all classes related). An example of a policy with totally ordered classes would be the government classification (unclassified, confidential, secret, top secret) of information, oac and the access class of its' users, sac, called the user's clearance. For such a lattice policy the system must insure that access to classified information is always confined to cleared users.

DISCRETIONARY POLICY involves checking an access control list (ACL). If the user requesting access is not included on the ACL then the access is not permitted. This allows users to specify who can access their files. This

policy really lies within the non-discretionary structure and provides further refinement. This policy would reflect the "need to know' rule of DOD.

There are many distinct system designs which correspond to the almost endless number of policies; however, the current state of the art allows a simple, uniform mechanism for nearly all practical policies. The implication is that the kernel designer does not have to concern himself with the particular security policy of a specific customer. He must, however, consider the two broad classes of policy: discretionary and non-discretionary.

2. Mathematical Model

A mathematical model[13] is a powerful design tool for formally translating the requirements of security policy into a precise representation of the behavior of the corresponding security kernel. The mathematical model is a finite state machine model that gives a set of rules of operation for making a state transition. If the system is initialized to a secure state, then the rules of operation guarantee that all subsequent states are secure. Previous research[14] has proven that security kernels whose design is based on mathematical models can be certified correct.

Two of the basic elements of the model are subjects and objects. The model defines types of accesses that a subject may have to an object. These access types are read and/or write. The state of the system with respect to

non-discretionary and discretionary security is represented by four sets (b, m, f, h). This design implements non-discretionary security policy in the kernel (sets b, f) and the discretionary policy in the supervisor (sets m, h). The folowing discussion pertains to non-discretionary security.

b - represents the current access relationships that exists between all subjects and objects. This set is represented by the segment descriptor list, viz., the contents of the hardware registers in the MMU (memory management unit).

f - gives the access class of all subjects and objects in the system. This set is distributed in this design: the process's access class is found in the active process table (APT) and the segments access class is in the active segment table (AST).

The desired properties of the system are then realized in the form of rules. These rules enforce the desired security policy by manipulating the sets which may or may not change the state of the system. If the state of the system is changed it must guarantee that the new state is secure.

The discretionary security policy is enforced in the supervisor. This design decision was made because of the lesser importance of "need to know" controls to the military, and to keep the kernel small for ease of verification.

The sets which are used to enforce the discretionary policy are m and h.

m - corresponds to an access matrix which represents the potential access of the subjects to objects (implements the "need to know" security policy). This set is represented by the access control list for the segment (object).

h - indicates how the objects are hierarchically organized in a directory tree structure. The hierarchical tree structure consists of nodes, leaves, and a root from which the tree eminates. The nodes represent a directory segment (list of attributes for other segments) and the leaves represent non-directory segments (data or procedure). A user is free to create either directory or non-directory segments. The ability to add directories implies that a user, if he chooses, can add to the overall system hierarchy a subtree of arbitrary depth.

3. Properties And Conditions

There are a few basic security properties which need to be considered:

SIMPLE SECURITY CONDITION- this condition addresses the problem of security compromise. If in set b all subjects have an access class greater than or equal to the access class of their objects, this condition is satisfied. This insures the subject only reads information at or below the class for which it is cleared.

CONFINEMENT - this property addresses potential

(rather than actual) security compromises. If all subjects could be trusted to perform in a proper manner (with respect to security), then this property would not be needed. The fact is that unless a program is proven to behave in a certain fashion as described by the mathematical model or formal specification, we cannot make any statements concerning its behavior. We must therefore make the assumption that the programs will attempt to violate security regulations. Subjects are therefore assumed to be untrustworthy. The potential for a security compromise occurs when a subject has simultaneous read access which is at class a and write access at class b (class a >class b). For example, the potential for compromise is realized if two events occur: (1) the subject reads secret information from the secret object and writes it into the unclassified object. (2) a second subject whose access class is unclassified gains access to this (nominally unclassified) object and reads the secret information. There are two ways of preventing this type of situation from occurring: high water mark and confinement property.

High Water Mark - upgrade the class of the file to the highest class requested. This solution, while technically correct, would over classify information so that it would not be available to normally cleared subjects.

Confinement Property (*-Property) - this property requires that all objects to which a subject has write access have the same access class as the subject and that

all objects to which it has read access have an access class less than or equal to the access class of the subject. Since a subject will always have write access to some object if it is to perform a computation, we define the current access class to be that class at which the subject wishes to have write access. Since all subjects are assumed untrustworthy with respect to security requirements, the confinement property eliminates the certification requirement outside the security kernel. This eliminates the immense job of certifying the supervisor and the user programs. This property is enforced in the kernel by not allowing any subject write access to an object with a lower access class.

COMPATIBILITY PROPERTY - If an object in the hierarchical structure is inferior (child) to an object (parent) and the access class of the parent is greater than that of the child, then a subject with an access class the same as the child can never access that information since it can not access the access control list which is kept in the parent. In order to avoid this problem we introduce the concept of "compatibility". A hierarchy is compatible if access classes are non-decreasing as one moves down the hierarchy from the root. The access class of an object in the hierarchy must always be greater than or equal to the access class of its parent. Since the root has no parent its security attributes are implied (viz., are the "lowest" of any object). In this design compatibility is enforced in the kernel, but not in the traditional sense of enforcing the

access relationship of the parent/child hierarchical structure. There is no hierarchical structure in the kernel. When the segment is created the compatibility is implicitly enforced before the request is allowed.

The reference monitor is an abstraction of the hardware and software mechanisms that mediate all attempts by subjects to access objects. The decision to permit or deny access is determined by the security kernel. The mathematical model is an interpretation of the reference monitor abstraction and describes the behavior of a secure system in terms of four component data bases (b, m, f, h) and rules of operation. These rules specify how the data base may be changed, they represent an "authorize" operation. The security kernel can only allow subjects to access objects as permitted by its representation of the model's set b. The data base of the security kernel must correspond to the model's data base and can only change as permitted by the model's rules.

The reference monitor of a physical computer system is realized by a combination of software and hardware. The portion required in software depends on the capabilities and limitations of the hardware. There may be objects to which the hardware can not properly control access and there may be alternative representations of the same security state. Either one of these situations require a kernel function that does not change the security state. In the former case there would be one or more functions to permit interpretive

access to an object; in the latter there would be functions for changing the representations of the security state without changing the actual state.

Thus the functions of the security kernel software[2] fall into three classes that correspond to the fundamental operations of authorize, access, and null: (1) functions that correspond to the rules of the model, thus changing the security state; (2) functions that implement a part of the reference monitor by allowing interpretive access to objects as permitted by the current security state, thus complementing the hardware access controls and (3) functions that change the representation of the current security state.

4. Segmentation

The mathematical model addresses abstract subjects and objects. In this design subjects are the processes and the principal information objects are segments. Processes (subjects) can only access segments (objects) as permitted by the access controls. Every segment has associated with it logical attributes (access class, size, read/write permission) which are made visible at the time of actual reference to the information. By including access control as part of the logical attributes, a way to control access to the information in the system has been provided. Only "authorized" accesses are allowed.

Segmentation provides the mechanism so that all

online information stored in the system is directly addressable by a processor and hence available for direct reference by any computation. A basic advantage of direct addressability is that users can physically share a single copy. A concern which arises from sharing is that information may be passed illegally between users. This is prevented by the enforcement of the confinement property and the simple security condition. The copying of data is no longer mandatory as many users can share a single copy with controlled access.

5. Hardware Requirements

There are no absolute hardware requirements for secure computer systems, any hardware is theoretically acceptible. Given the current state of the technology, however, certain hardware features are essential if we are to build efficient secure systems[2]. These essential features reduce and simplify the software portion of the security kernel. Reduction and simplification of software at the expense of additional hardware is necessary because producing proveably correct software and hardware in the security kernel is a necessity to achieve computer security.

One of the essential features is support for a segmented memory. Segmentation allows all information in the system to be stored in one type of object, the segment. Having to support only a single object type simplifies the kernel. Segmentation allows all information in the system to

be compartmentallized into individual packages called segments. Every segment has associated with it access controls as previously mentioned. Only authorized accesses as delineated in the access control list and allowed by the access class are permitted. The address of information is composed of two parts (segment #, offset). It is necessary to efficiently resolve the two dimensional address into an absolute address, therefore segmentation should be implemented in hardware.

The other essential hardware feature is multiple execution domains. This feature is used in most contemporary systems to protect operating systems from applications programs. Strictly speaking only two execution domains are necessary (one for the kernel and one for everything else), but in practice it will still be desireable to continue to protect the operating system from applications software so three domains (kernel, supervisor, user) will be used in this design.

III. DESIGN

A. DESIGN TECHNIQUES

When designing an operating system there are several approaches to consider: top down, bottom up and middle out. Although most designs begin as top down or bottom up they generally end up as middle out. In the design there are several design choices available to the designer. In some cases a certain design choice will preclude the ability to utilize a specific design later on in the system design, while in other cases a specific design choice could be a driving force to dictate other design choices. For example in the SYSTEM the design choice was made to keep the kernel relatively small to reduce the verification process. This particular choice became a heavily weighted factor when, for example, deciding where to support the demand memory management which ended up in the supervisor. Following are some of the design techniques that contributed to the SYSTEM.

1. Resource Virtualization

By using virtual processors and virtual memory throughout the upper levels of the design, most of the design is independent of the physical configuration. The SYSTEM provides the virtual to real binding in the kernel. This permits changing the configuration to meet user or

maintenance requirements without major changes to the system. Since the processes are assigned virtual processors there is no effect on the user when real processors are added or deleted (except for the change in performance). Of particular interest was the ability to add and delete processors to the SYSTEM. More important was to develop a design that allowed good capacity growth with the addition of processors. In general, configuration independence implies that the hardware (processors, memory and peripherals) can be reconfigured without causing any problems visible to the user.

2. Distributed System

The SYSTEM is distributed logically and physically. Logically, portions of the operating system are distributed within the address space of the users process within the supervisor and kernel domains. The use of domains permits the process to maintain its security attributes while interacting with the operating system.

The physical distribution of segments among the individual local memories provides performance (provides high speed memory access and limits BUS contention). The physical distribution allows the tradeoff of memory (viz., multiple copies) for performance. Although one of the potential benefits of segmentation is sharing of pure procedures the choice was made to disregard this benefit when possible (no user has write access). This allows the

segment (viz., a copy) to reside in local memory to reduce BUS contention. The initial hypothesis is that the memory wasted (much of it possibly ROM) is a small price to pay to allow performance to grow well with the addition of processors. This addresses the problem that in typical multiprocessor systems capacity scales poorly because of increase load on the BUS. However, this choice is not fundamental to the design and could be changed to eliminate multiple copies.

Similarly for processors, processing is distributed to processors to eliminate the dependency on a single controlling unit. The system wide data bases are kept in global memory providing access to all processors.

3. Multiple Protection Domains

The foremost consideration in the design of the SYSTEM was security. This is acheived by use of the security kernel technology, and segmentation provides one of the keys to providing security within the system. The set of segments that are accessible is defined as a domain. The conventional two state system does not provide the desired support for a secure system. For this reason the 2-state (and associated 2 domains) is generalized to a hierarchical n-domain system[6]. In the design of the SYSTEM (a minimum of) 3-domains were considered adequate - user, supervisor and kernel. In addition, the design permits that, based on user application, a number of user domains could be supported.

Each domain is in concept similar to a ring[6]. The authorized access of a process is determined by the current ring of execution. The access within the different rings form a set of nested domains. Ring Ø (kernel) is the largest set and ring n-1 is the smallest.

The ring structure with the associated controls provides a means for regulating the information that passes between domains (rings). Cross-ring calls and parameter passing are well defined[15]. When the proper controls are used they allow outer rings to make requests to inner rings, but also protect the inner rings from unintentional or intentional tampering. The ring structure when combined with segmentation provides mechanism for the design of an effective secure system by protecting the secure kernel.

4. Multiprocessing

The process structure provides the essentials for parallel processing: support for a set of assynchronous processes that can communicate with each other. Parallel processing does not require a multi-processor environment. However, in a multi-processor environment parallel processing can provide faster completion of a job.

There are many applications for parallel processing within tactical as well as non-tactical systems. Whenever a job depends on a mixture of asynchronous and synchronous tasks and time is a factor, parallel processing is a possible solution to getting the job done in the allocated

time. By using several processors working on the same job, each doing seperate tasks, the overall time required to do the job can be reduced (provided the job has been structured into explicit processes). In microprocessors where processors are relatively inexpensive and slow, parallel processing may be the answer to keeping the cost down while still being able to complete the job in the required time. The above discussion provides some of the major reasons why the SYSTEM was designed to support parallel processing on multiple processors.

5. 'Cache' Memory Strategy

A cache memory is generally thought of as a small amount of high speed memory that is utilized with a large low speed main memory in a system to construct a memory system that appears to be a larger high speed memory. This appearance of a high speed memory is generally possible as a result of locality of reference[16,p.301].

In a multiprocessor environment, where each processor has its own cache memory, problems arise when accessing shared memory. The main problem being that shared, writable memory cannot be put in a cache. Segmentation allows the assignment of attributes to segments, which provides a way to identify cacheable segments (those segments that are not writable and shared).

In a multi-microprocessor system where BUS contention can become a problem a cache memory strategy

could be quite effective in reducing the number of requests to the main memory, even though the cache and shared memory are the same speed. The main advantage is avoiding access to the system BUS rather than the increase in speed of the actual memory access. The SYSTEM uses the strategy of a cache in the form of a local memory per processor. Now rather than being a copy of what is in global memory the local memory (cache) becomes the place where the data is stored instead of global memory (note that with a cache, global memory need not contain a copy while the information is in the cache).

Each processor has its own local memory which is relatively large in size where cacheable segments are stored. This means that large blocks of data will be moved when a process is removed from one processor and (subsequently) loaded on another processor. In addition a global memory is utilized for shared writable segments (unencacheable segments). Segmentation allows the SYSTEM to utilize the concept of caches and main memory but in the form of local and global memory. The overall reason is the same (speed up memory access), but in the SYSTEM this is achieved by reducing the BUS contention through directing most access to local memory.

6. Multiprogramming

In a system where there are more processes than processors there must be a means of switching processors

from process to process. Some reasons for switching process are: current process completes, a higher priority process is ready, current process is blocked, or current process is waiting I/O. Whatever the reason for switching, there are certain things that must be done in performing the switch: first, save the address space of the old process as well as the current execution point represented by a portion of the processor state, and secondly, reloading the address space and previous execution point of the new process. The process switch must occur in a specific sequence to insure the new process resumes execution at the same point and in the same logical state as when it was previously switched. In the SYSTEM re-establishing the local memory to its previous state becomes part of the process switch (when switching user processes).

Because of the overhead (unloading and loading all the MMU registers) associated with process switches, provisions are included to make the processes semi-dedicated to a processor and thus make the requirement for memory switches infrequent. In order to make the process switch totally hidden outside the kernel, the segments that were in memory the last time the process was executing must be loaded in memory prior to allowing the process to resume execution. The lack of a "DBR"[6,p.12] is a problem, but saving copies of the MMU, that can be reloaded when required reduces the severity of the problem.

7. Family of Operating Systems

The design in this thesis is not really for a single operating system, but rather for a whole family of operating systems. For any specific system the family member chosen depends on the functions required. A tactical system which is static in nature does not require many of the user services supported by the SYSTEM. For this reason the family member that consists of only the kernel could be the specific operating system chosen for a tactical system. A general purpose time sharing system, on the other hand, is very dynamic in nature, utilizing large address spaces, variable number of users, etc. The family member that supports dynamic linking, a hierarchical file system and demand memory management could be the specific operating system for the general purpose time sharing system.

Operating system sub-setting refers to the ability to form meaningful sub-sets of an operating system. In the design of the SYSTEM a sub-setting capability was one of the goals. The structure is such that many of the services provided by the SYSTEM can be eliminated without effecting the usefulness of the remaining system. That is the SYSTEM can be tailored to fit a number of specific requirements. This is made possible primarily by utilizing a loop free structure[9] within the design. For explanation purposes consider the operating system to be composed of modules. In a loop free structure the dependency is inward or downward (toward the hardware), depending on your point of view. A module only depends on another module at a lower level.

Requiring a loop free dependency structure allows system correctness to be established one module at a time. Modifying a module would only effect the modules above which depend on it.

The design choice to keep the kernel relatively small and put the common user services in the supervisor lends itself to sub-setting. The security kernel would not be changed in any of the sub-sets and thus would not require re-verification. The supervisor supported services (dynamic linking, discretionary security, demand memory management, hierarchical file system) could be removed to meet the needs of the specific use of the system. This makes the sub-sets of the SYSTEM suitable for tactical application, where there is generally no need for demand memory management or dynamic linking (static environment), as well as for general purpose application where all the features can be utilized. It should be noted that any of these meaningful sub-sets would be a secure system since the kernel remains unchanged in every sub-set. Sub-sets of the kernel can also be constructed; however, this would require reverification of the kernel.

8. Levels Of Abstraction

Abstraction is a way of avoiding complexity and a tool by which a finite piece of reasoning can cover a myriad of cases[17]. The purpose of abstracting is not to be vague, but to create a semantic level in which one can be

absolutely precise. Levels of abstraction have been demonstrated to be a powerful design methodology for complex systems. In general, the use of levels of abstraction leads to a better design with greater clarity and fewer errors. A level is defined not only by the abstraction that it supports (for example, a segmented virtual memory) but also by the resources employed to realize that abstraction. Lower levels (closer to the machine) are not aware of the abstractions or resources of higher levels; higher levels may apply the resources of lower levels only by appealing to the functions of the lower levels. This pair of restrictions reduces the number of interactions among parts of a system and makes them more explicit.

Each level of abstraction creates a virtual machine environment. Programs above some level do not need to know how the virtual machine of that level is implemented. For example, if a level of abstraction creates sequential processes and multiplexes one or more hardware processors among them, then at higher levels the number of physical processors in the system is not important. By the rules of abstraction calls to a procedure at a different level must always be made in a downward direction and the corresponding return in the upward direction. Note that at least two of the levels (kernel and supervisor) define virtual machines with rigidly enforced (via hardware) invokation of "extended instruction", i.e. the kernel and supervisor calls.

B. PROPOSED DESIGN

The SYSTEM is composed of two parts, the supervisor and the kernel. The supervisor provides operating system services while the kernel manages physical resources. This division also contributes to the ability to sub-set without affecting the kernel. The supervisor, which consists of procedures, is distributed and exists within the supervisor domain of each user process. The kernel is made up of both procedures and system processes. The procedures are part of the distributed operating system and exist within the kernel domain of each user process. The system processes are not distributed but are separate processes.

1. Notation

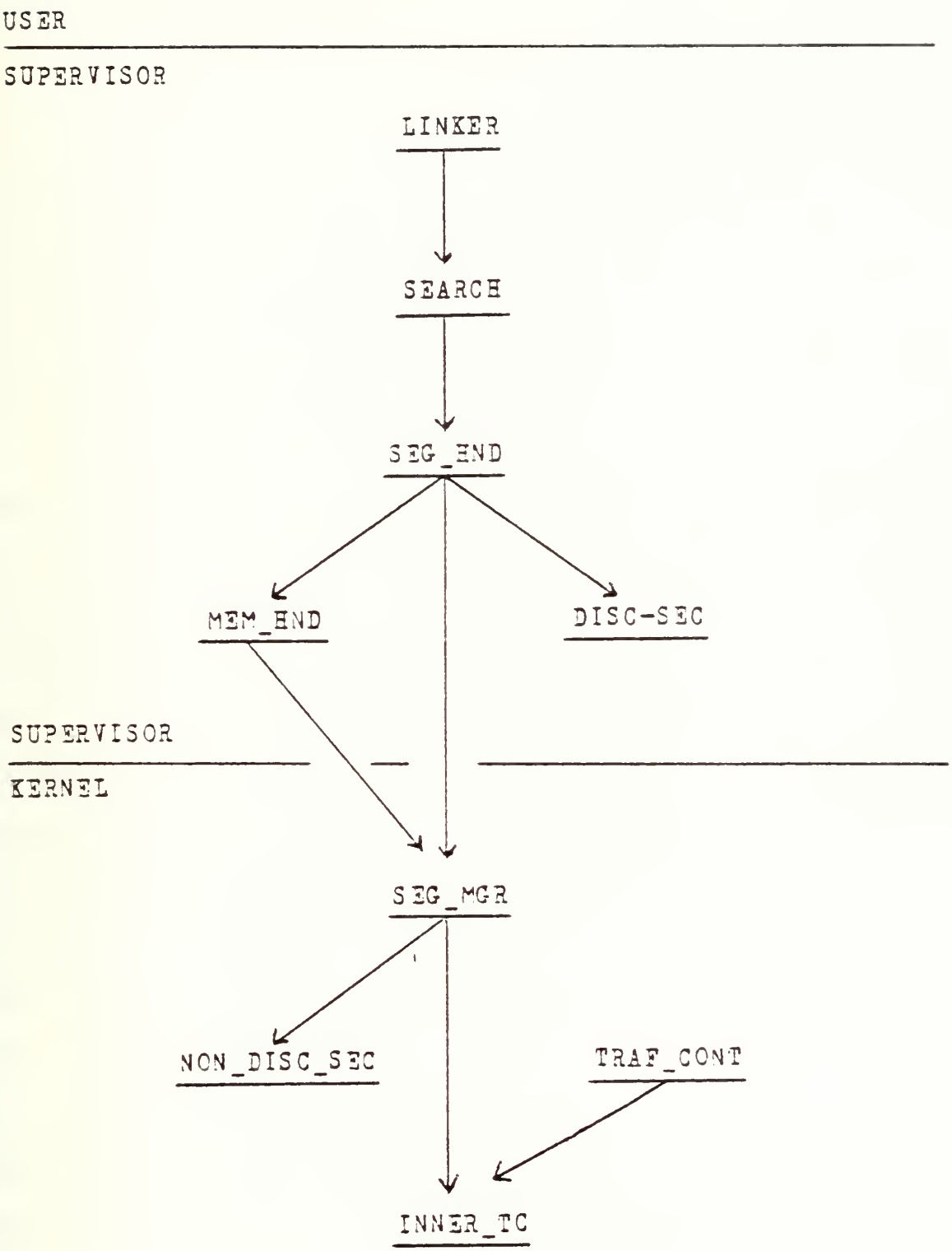
The following is an explanation of the notation used in the following discussions. When a CALL is used the name of the module is given followed by the parameters within parenthesis. When a name in quotes appears as the first parameter in the parantheses it is used to specify the entry within the module. For example CALL INNER_TC('UNLOAD', SEGMENT_#, WRITTEN) the module name is INNER_TC, 'UNLOAD' specifies the entry point and SEGMENT_# and WRITTEN are the parameters. When a SIGNAL is used the first name in quotes specifies the process for whom the signal is intended, the second name in quotes (optional) specifies the specific function requested of that process and the remaining names represent parameters. For example SIGNAL('MEMORY_MANAGER',

'OUT', SEGMENT_#, WRITTEN) the signal is meant for the memory manager process, 'OUT' is the requested function and SEGMENT_# and WRITTEN are parameters. WAIT is used when a process cannot continue execution until it receives a signal from another process. WAIT(PROCESS_ID, MSG). The return parameters PROCESS_ID and MSG are used to indicate the process that sent the signal and the message sent. It should be noted that the above notation is only used to simplify the understanding of what is happening. In an actual implementation the parameters need not be passed in precisely this fashion.

2. System Overview

The following is an overview of the SYSTEM's modules and processes and how they function. Figure 2 represents the modules that exist in the distributed supervisor and the distributed kernel. The levels are used to indicate the dependencies that exist between these modules. The supervisor is made up of four levels of abstraction. It should be noted that all data within the supervisor is per process.

The linker, a level 1 module called LINKER, exists in a segmented virtual memory and provides the mechanisms of dynamic linking. He is invoked by CALL LINKER(SYMBOLIC_NAME). It should be noted that the call could be by link fault as in MULTICS[6]. The linker keeps track of snapped links in the linkage segment (figure 3).



SYSTEM LEVELS

FIGURE 2

The linker utilizes the CALL SEARCH(SYMBOLIC_NAME, SEGMENT_#) to obtain the segment number for unsnapped links.

The searcher, a level 2 module called SEARCH, is invoked by SEARCH(SYMBOLIC_NAME, SEGMENT_#) and is required to return the segment number of the segment specified by the symbolic name. By applying the 'search rules' the symbolic name is converted to a path name in the hierarchical file system. The searcher gets the desired segment number by the CALL SEG_END(PATH_NAME, SEGMENT_#).

The segment handler, a level 3 module called SEG_END, is invoked by CALL SEG_END(PATH_NAME, SEGMENT_#) and is responsible for returning the appropriate segment number. The segment handler utilizes the Segment Table (figure 4) as its data base. To maintain the data base he uses the CALL SEG_MGR('MAKE_KNOWN', PAR_SEG_#, ENTRY_#, ACCESS, SEGMENT_#, SIZE) to the kernel to obtain a segment number for a segment and the CALL DISC_SEC(SEGMENT_#, ENTRY_#, ACCESS) to determine the authorized access (discretionary). The segment handler is also invoked by the virtual faults, SEG_END('SEG_FAULT', SEGMENT_#) and SEG_END('MEM_FAULT', SEGMENT_#). The 'SEG_FAULT' is a discretionary security access check and is handled by a CALL DISC_SEC(SEGMENT_#, ENTRY_#). The 'MEM_FAULT' is a request to bring a segment into memory and is handled by a CALL MEM_END(SEGMENT_#, SIZE).

The memory handler and discretionary security, level 4 modules called MEM_END and DISC_SEC respectively, are

SYMBOLIC_NAME	SEGMENT_#	OFFSET
TEST1	4	Ø
TEST2	5	Ø
THESIS	6	Ø

LINKAGE SEGMENT (PER SEGMENT)

FIGURE 3

SEGMENT_#	DISC-SEC ACCESS	SIZE	PARENT SEGMENT_#	ENTRY #

SEGMENT TABLE

FIGURE 4

FREE		ALLOCATED		
BASE	SIZE	BASE	SIZE	SEGMENT_#
BASE	SIZE	BASE	SIZE	SEGMENT_#
BASE	SIZE			

MEMORY MAP (LOCAL)

FIGURE 5

invoked by MEM_END(SEGMENT_#, SIZE) and DISC_SEC(SEGMENT_#, ENTRY_#, ACCESS) respectively. The memory handler provides the dynamic memory management utilizing the Memory Map data base (figure 5). The memory handler uses the CALL SEG_MGR('SWAP_IN', SEGMENT_#, BASE_ADDRESS) in the kernel to bring a segment into memory and the CALL SEG_MGR('SWAP_OUT', SEGMENT_#) to remove a segment. The discretionary security checks the access control lists to determine the authorized access of the process (discretionary).

The distributed kernel is composed of three levels. The segment manager, a kernel level 1 module called SEG_MGR, is invoked by the CALL SEG_MGR('MAKE_KNOWN', PAR_SEG_#, ENTRY_#, ACCESS, SEGMENT_#), CALL SEG_MGR('SWAP_IN', SEGMENT_#, BASE_ADDRESS) and CALL SEG_MGR('SWAP_OUT', SEGMENT_#). The segment manager maintains the Known Segment Table (figure 6) as a per process data base. The segment manager determines allowable access by the CALL NON_DISC_SEC(UNIQUE_ID, ACCESS) and assigns segment numbers by the CALL INNER_TC('ASSIGN', SEGMENT_#, ACCESS). The segment manager brings segments into memory by SIGNAL('MEMORY_MANAGER', 'IN', SEGMENT_#, UNIQUE_ID, BASE_ADDRESS) and removes segments from memory by SIGNAL('MEMORY_MANAGER', 'OUT', SEGMENT_#).

The non-discretionary security, a kernel level 2 module called NON_DISC_SEC, is responsible for determining the authorized access for a given segment. Non-discretionary security is invoked by the CALL NON_DISC_SEC(UNIQUE_ID,

UNIQUE ID	SEGMENT_#

KNOWN SEGMENT TABLE (KST) ENTRY

FIGURE 6

PROCESS_ID	STATE	AFFINITY	PRIORITY	LOC_EX_STATE	VIRTUAL PROCESSOR_#

ACTIVE PROCESS TABLE ENTRY

FIGURE 7

VIRTUAL PROCESSOR_#	VIRTUAL PROCESSOR PRIORITY	PROCESS ID
1	3	
2	7	
3	4	
4	10	

PROCESSOR TABLE

FIGURE 8

ACCESS).

The traffic controller, a kernel level 2 module called TRAFFIC_CONT, is responsible for multiplexing user processes to virtual processors. The traffic controller utilizes the Active Process Table (figure 7) as its data base. traffic controller is invoked by the CALL TRAFFIC_CONT('BLOCK', MSG, WAKING_ID) and CALL TRAFFIC_CONT('WAKEUP', PROCESS_ID, MSG). The traffic controller uses the SIGNAL('MEMORY_MANAGER', 'LOAD', VIRT_MEM_MAP) and SIGNAL ('MEMORY_MANAGER', 'UNLOAD', WRIT_BIT_MAP) to load and unload the processes' segments in memory on the virtual processors. The traffic controller uses the CALL INNER_TC('LOAD_MMU', PROCESS_ID) AND CALL INNER_TC('UNLOAD_MMU') to load or unload the memory management registers of the virtual processors. The traffic controller uses the CALL INNER_TC('IDLE') to remove a virtual processor from contention for rescources. Actually the virtual processor is assigned the lowest priority available and the idle process is loaded.

The inner traffic controller, a kernel level 3 module called INNER_TC, provides the multiplexing of virtual processors to real processors. The inner traffic controller uses the Processor Table (figure 8) as its data base.

The non-distributed kernel consists of two system processes. The memory manager process maintains the Active Segment Table (figure 9) and Global Memory Map (figure 10) as data bases. Basically it loads segments into memory. The

UNIQUE_ID	GLOBAL ADDRESS	WRITTEN BIT	PROCESSOR BIT MAP	CONNECTED PROCESSES	WRITEABLE BIT

AST GLOBAL

VIRTUAL PROCESSOR_ID	SEGMENT_#	UNIQUE_ID	ACCESS	ABS ADDRESS

AST LOCAL (PER PROCESSOR)

ACTIVE SEGMENT TABLE

FIGURE 9

FREE		ALLOCATED		
BASE	SIZE	BASE	SIZE	SEGMENT_#
BASE	SIZE	BASE	SIZE	SEGMENT_#
		BASE	SIZE	SEGMENT_#

GLOBAL MEMORY MAP

FIGURE 10

memory manager process is responsible for putting segments in local/global memory based on user's access.

The I/O manager process processes all the external I/O, this includes I/O to and from the user terminals. The terminals can be thought of as being hard wired. Specific terminals have specific access classes; therefore no kernel passwords are required to determine access class.

The next three sections provide a detailed discussion of the design.

3. Supervisor

The supervisor can be invoked by the following external (user) calls:

```
SUP_CREATE_SEGMENT(ACCESS_CLASS,SIZE)
SUP_DELETE_SEGMENT(SEGMENT_#)
LINKER(SYMBOLIC_NAME)
SUP_BLOCK(MSG)
SUP_WAKEUP(PROCESS_ID,MSG)
SUP_CREATE_PROCESS(PROCESS_ID, ADDRESS_SPACE)
SUP_DESTROY_PROCESS(PROCESS_ID)
```

a. Linker (Supervisor)

The linker exists in a segmented virtual memory environment. It is only aware of symbolic names and segment numbers. The choice was made to provide dynamic linking and not assign segment numbers to segments at compile or load time; therefore there is a requirement to resolve external references at run time. In general it is the linker's job to

intervene on a procedure's external references and direct the reference to the appropriate segment. To accomplish this the linker utilizes a "linkage segment" (each process has a linkage segment). The linkage segment contains an entry for each segment known to the process.

Each external reference results in a call to the linker with a parameter that on first reference permits finding the symbolic name of the desired segment.

LINKER(SYMBOLIC_NAME) The linker searches for the entry corresponding to the symbolic name. If found it transfers to the segment number and offset specified in the linkage segment. If not found (first reference) it must first determine the segment number and offset. To obtain the segment number the linker calls the searcher passing as a parameter the symbolic name. SEARCH(SYMBOLIC_NAME, SEGMENT_#) The parameter returned is the segment number. The linker completes the entry in the linkage segment and transfers control to the desired segment.

b. Searcher (Supervisor)

The searcher is aware of the hierarchical file system and a set of search rules. It is invoked by SEARCH(SYMBOLIC_NAME, SEGMENT_#). The searcher has the task of resolving a symbolic name into a path name. The searcher recieves as a parameter a symbolic name which is processed and eventually the segment number of the symbolically named segment is returned. To accomplish this the searcher applies the 'search rules'[6]. The search rules are a list of path

names and a simple technique that convert the symbolic name to a path name (note that this is independent of security). The searcher utilizes a calling directory and working directory [6,p.230]. Once the path name is determined the searcher calls the segment handler passing the path name as a parameter. SEG_END(PATH_NAME, SEGMENT_#) The parameter returned is the segment number. The searcher returns passing the segment number as a parameter to the linker.

c. Segment Handler (Supervisor)

The segment handler understands the hierarchical file system, parent, entry number, access control lists, and segment numbers. The segment handler deals with virtual segment faults (access checks) and virtual memory faults. He is involked by the call SEG_END(PATH_NAME, SEGMENT_#). The segment handler gets assistance in performing his tasks by utilizing the following calls: MEM_END(SEGMENT_#, SIZE) to request a segment be put in virtual memory, DISC_SEC(SEGMENT_#, ENTRY_#) a function to determine the authorized access (discretionary security) to a segment, SEG_MGR('MAKE_KNOWN', PAR_SEG_#, ENTRY_#, ACCESS, SEGMENT_#) a kernel call used to determine the segment number and size of the segment indicated by the parent segment number and entry number.

The segment handler maintains a segment table with information that is necessary to control segments at the supervisor level (figure 4). The segment number is unique within the process. Parent segment number is the

segment number of the parent and entry number is the entry within the parent for the segment. Access is that access authorized by the discretionary security policy. Size is the memory required by the segment. The segment handler is required to convert path names to segment numbers as well as to handle virtual segment faults (discretionary security checks) and virtual memory faults. To accomplish these tasks the segment handler has three entry points: SEG_END, MEM_FAULT and SEG_FAULT.

SEG_END(PATH_NAME, SEGMENT_#) The segment handler receives as a parameter the path name of the desired segment. One of the design characteristics of the hierarchical file system is that access to a segment requires read access to every segment on the path of the segment. One by one the segments on the path name must be made known and the access established. To do this a recursive algorithm can be utilized that will process each entry within the path name until the path name is resolved. The segment number assigned to the desired segment is returned.

SEG_END('MEM_FAULT', SEGMENT_#) A virtual memory fault is utilized to support the dynamic memory management outside the kernel. When a segment that is not in memory is referenced a virtual memory fault (hardware initiated, the kernel provides the software interpretation of the fault and provides a transfer vector to the supervisor) is generated to the segment handler. The segment handler uses the Segment

Table to determine the SEGMENT_# and the SIZE of the segment. The memory handler is called, MEM_END(SEGMENT_#, SIZE).

SEG_END('SEG_FAULT', SEGMENT_#) A virtual segment fault is used to tell the supervisor that the ACL for the segment referenced has been changed since the last time the segment had been referenced. The segment handler must re-establish the discretionary security. This is done by checking the Segment Table for the parent's segment number and entry number, calling DISC_SEC(SEGMENT_#, ENTRY_#, ACCESS), check the new access, update the Segment Table and return.

d. Memory Handler (Supervisor)

It is the job of the memory handler to provide the dynamic memory management within a fixed size linear virtual memory. The memory handler utilizes two kernel calls 'SWAP_IN' and 'SWAP_OUT' to perform his tasks. SEG_MGR('SWAP_IN', SEGMENT_#, BASE_ADDRESS) is used to request that a segment be brought into memory. SEG_MGR('SWAP_OUT', SEGMENT_#) is used to remove a segment from memory.

The memory handler is tasked by the segment handler to put a segment into memory and provided with the SEGMENT_# and SIZE of this segment. The data base utilized is a Memory Map (figure 5) which indicates free areas and allocated areas. Each process has a memory map which is used to keep track of the virtual memory allocated to the

process.

To provide the demand memory management there are many suitable algorithms[16,p.155]. First fit, best fit and worst fit are among the possible choices for allocating free areas. A least recently used algorithm is generally used for deallocating memory. The used bit is available to provide information to the dealocation scheme. The CALL INNER_TC('GET_USED_BITS', USED_BITS) returns an array of the status of all the used bits. The CALL INNER_TC('SET_USED_BITS', USED_BITS) provides an array of the desired value of the used bits. This provides the mechanism for an approximating efficient Least Recently Used algorithm for dealocation[16]. Allocated areas (figure 5) are identified by (SEGMENT_#, BASE_ADDRESS, SIZE). When tasked, the memory handler searches for a free area large enough for the segment. If there is no free area large enough, the memory handler must utilize the CALL SEG_MGR('SWAP_OUT', SEGMENT_#) to establish a large enough free area. The memory map is updated and the CALL SEG_MGR('SWAP_IN', SEGMENT_#, BASE_ADDRESS) is generated. The memory map is updated and the memory handler returns.

e. Discretionary Security (Supervisor)

This module is only aware of access control lists (figure 11) and how to search one to determine the access to be given the current process. The input parameter is the segment number (of the directory) and entry number of the ACL for the desired segment. The discretionary security

searches the ACL for the PROCESS_ID of the calling process and thereby determines the access, which is returned.

4. Distributed Kernel

There is a gate mechanism (domain change) through which all kernel and supervisor calls pass. Checks are made to determine proper (complete) parameters and the call is directed to the proper module. The kernel is the "priviledged mode" and can execute priviledged instructions. Calls coming from outside the kernel are:

MAKE_KNOWN(PAR-SEG_#, ENTRY_#, ACCESS, SEGMENT_#)

SWAP_IN(SEGMENT_#, BASE_ADDRESS)

SWAP_OUT(SEGMENT_#)

SET_SEG_FAULT(SEGMENT_#)

BLOCK(MSG,WAKING_ID)

WAKEUP(PROCESS_ID, MSG)

CREATE_PROCESS(PROCESS_ID, ADDRESS_SPACE)

START_PROCESS(PROCESS_ID, EXECUTION_POINT)

STOP_PROCESS(PROCESS_ID)

DESTROY_PROCESS(PROCESS_ID)

CREATE_SEGMENT(PAR_SEG_#, ENTRY_#, ACCESS_CLASS, SIZE)

DELETE_SEGMENT(UNIQUE_ID)

INNER_TC('GET_USED_BITS', USED_BITS)

INNER_TC('SET_USED_BITS', USED_BITS)

a. Segment Manager (Kernel)

The segment manager's environment is a segmented

physical memory. The segment manager assigns segment numbers and is responsible for maintaining the status of all segments known to a process. The segment manager's primary data base is the Known Segment Table (KST) (figure 6). The unique_ID is a unique, system wide identifier assigned to each segment. They are assigned from an available list of integers (can be reused when a segment is deleted). Each segment also has an alias that is the unique_ID of and the entry number in its parent. This provides a means of determining the unique_ID of a segment from the segment number of and entry number in the parent.

It should be noted that the reason for the alias is to prevent the unique_ID from leaving the kernel. The alias chosen is derivable from information known to the supervisor, because it relates to the hierarchical file system. This information is per process and not system wide in nature. Although the hierarchical structure of the file system can be derived from the kernel's alias data base, the contention is that the file system in the kernel is a flat one. This method also eliminates the confinement problem. The kernel only requires that the access class of a segment, when created must be at or above the access class of the process creating the segment.

The segment manager can be invoked by several calls:

SEG_MGR('MAKE_KNOWN', PAR_SEG_#, ENTRY_#, ACCESS, SEGMENT_#)

SEG_MGR('SET_SEG_FAULT', SEGMENT_#)

SEG_MGR('SWAP_IN', SEGMENT_#, BASE_ADDRESS)

SEG_MGR('SWAP_OUT', SEGMENT_#)

The CALL SEG_MGR('MAKE_KNOWN', PAR_SEG_#, ENTRY_#, ACCESS, SEGMENT_#). The task is to assign a segment number to the segment specified. PAR_SEG_# and ENTRY_# are the segment number of the parent directory and the entry within that directory. The parent segment number is used to determine the unique_ID of the parent from the KST and this combined with the entry number forms an alias for the desired segment. The segment manager searches the KST to determine if the segment has already been assigned a segment number (already known). If this is the case the segment number already assigned is returned. If the segment is not known then a KST entry must be made. The procedure is as follows: use the PAR_SEG_# and the KST to determine the unique_ID of the parent. Combine the unique_ID of the parent and the entry number to derive the alias of the segment. Use the alias to determine the unique_ID of the desired segment from the alias table (figure 12). CALL NON_DISC_SEC(UNIQUE_ID, ACCESS) to determine the authorized access. The access granted is the desired access or the authorized access, whichever is less. Assign a segment number. Fill in KST entry. CALL INNER_TC('ADD_SEG', SEGMENT_#, ACCESS). Return assigned segment number.

The CALL SEG_MGR('SET_SEG_FAULT', SEGMENT_#). This call is used when the access control list for a segment

FILE	ACL
THESIS	'OCONNELL'(ALL ACCESS), 'RICHARDSON'(ALL ACCESS) 'JONES'(READ ACCESS), 'ALL_OTHERS'(NO ACCESS)

ACCESS CONTROL LIST

FIGURE 11

UNIQUE ID	ALIAS	
	(PARENT UNIQUE_ID)	(ENTRY_#)
15	7	3
13	12	2
7	1	1
12	7	4

ALIAS TABLE

FIGURE 12

MACHINE REGISTERS		
SOFTWARE FAULTS	ACCESS	RELATIVE BASE ADDRESS

LOC_EX_STATE

FIGURE 13

is changed. The segment manager determines the unique_ID of the segment specified and does a SIGNAL('MEMORY_MANAGER', 'SET_SEG_FAULT', UNIQUE_ID).

The CALL SEG_MGR('SWAP_IN', SEGMENT_#, BASE_ADDRESS). A request to load the specified segment into memory at the indicated base address (relative). The segment manager locates the appropriate KST entry and does a SIGNAL('MEMORY_MANAGER', 'IN', SEGMENT_#, UNIQUE_ID, BASE_ADDRESS) and a WAIT(PROCESS_ID, ABS_ADD, BOUND). The memory manager process loads the segment in memory and returns the absolute address and bound of the segment. The segment manager notifies the inner traffic controller of the update in segment information CALL INNER_TC('LOAD', SEGMENT_#, ABS_ADD, BOUND). The segment manager returns.

The CALL SEG_MGR('SWAP_OUT', SEGMENT_#). The segment manager is tasked with removing the segment from memory. He does a CALL INNER_TC('UNLOAD', SEGMENT_#, WRITTEN) to obtain the value of the written bit and then to unload the segment from memory a SIGNAL('MEMORY_MANAGER', 'OUT', SEGMENT_#, WRITTEN), WAIT('MEMORY_MANAGER') and then returns.

b. Non-Discretionary Security (Kernel)

The purpose of the non-discretionary security is to enforce the non-discretionary security policy by checking the access class of the process against the access class of the desired segment. The access is determined as a result of this comparison. The non-discretionary security module is

invoked by the CALL NON_DISC_SEC(UNIQUE_ID). An algorithm is used for interpreting the lattice for comparing the access classes and determining the authorized access. The non-discretionary security module returns passing the access.

C. Traffic Controller (Kernel)

The job of the traffic controller is to schedule and control processes. The traffic controller utilizes an Active Process Table (system wide) (figure 7) and a Virtual Processor Table (figure 8) to maintain the necessary information about each process. Each virtual processor has a priority (this priority is used by the inner traffic controller when the virtual processors are multiplexed on the physical processors). PROCESS_ID is a unique identifier for each process, which can be mapped to the user. STATE refers to the present state of a process (ready, block, stop, run). AFFINITY is used to specify a binding of a process to a virtual processor either by virtue of dissimilar processor characteristics (strong) or the process has segments in local memory of a processor (weak). PRIORITY is used to determine a scheduling behavior. LOC_EX_STATE provides the means for keeping track of the execution state of the process and is a pointer to a storage area that contains information about the execution state (figure 13).

The traffic controller schedules the processes to run on virtual processors. There is a virtual processor for every loaded process. Each virtual processor has a low

priority process (IDLE) so that the processor is never stopped. The traffic controller provides the BLOCK and WAKEUP functions as a means of providing inter-process communication.

The traffic controller would have a priority driven scheduling algorithm to determine what process to schedule. This could be a simple first come first served algorithm or it could be a complex time sharing algorithm to dynamically change process priority. The method utilized in this thesis is that the traffic controller works on the premise of scheduling the ready process with the highest priority and the proper affinity whenever a virtual processor is available.

Whenever a process blocks itself it is in fact a call to the traffic controller. The traffic controller changes the state of the process to blocked, The traffic controller now has the option of reassigning the virtual processor to another user process or scheduling the idle process (CALL INNER_TC('IDLE')). In the latter case there is no loading or unloading of the process involved and this can be beneficial to control thrashing. Since there are other virtual processors competing for the processor the traffic controller scheduling algorithm will try to leave the process loaded. When the process is put back in the run state it will be in contention for the processor. If another process is to be assigned to the virtual processor then the old process must be unloaded. First the status of the

written bits are determined (CALL INNER_TC('WRITTEN_BITS')). The execution state of the old process is unloaded (CALL INNER_TC('UNLOAD_MMU', PROCESS_ID, LOC_EX_STATE)). SIGNAL('MEMORY_MANAGER', 'UNLOAD', WRIT_BIT_MAP) and WAIT('MEMORY_MANAGER', VIRT_MEM_MAP) are generated, the virtual memory map of the process is returned by the manager process process. The execution state and the virtual memory map of the old process are saved. Now the new process can be loaded. The virtual memory map of the new process is passed to the memory manager process, a SIGNAL('MEMORY_MANAGER', 'LOAD', VIRT_MEM_MAP) and WAIT('MEMORY_MANAGER', ABS_ADD_MAP) are generated. A map indicating the absolute address of the loaded segments is returned by the memory manager process. The execution state of the new process is loaded (CALL INNER_TC('LOAD', LOC_EX_STATE, ABS_ADD_MAP)). This completes the process of switching user processes on a virtual processor.

The TRAFFIC_CONT('WAKEUP'', PROCESS_ID) is also a call to the traffic controller. If the process specified by PROCESS_ID is in the blocked state the traffic controller puts that process in the ready state, he checks the priorities of the running processes and if there is a lower priority process in the run state the virtual processor it is running on is sent a pre-empt interupt CALL INNER_TC('PRE_EMPT_INT', VIRT_PRO_ID) and the traffic controller returns. The pre-empt interupt forces the pre-empted virtual processor to transfer control to the

traffic controller. The traffic controller puts this process in the ready state and then schedules the highest priority process, subject to affinity, as indicated above. If the idle process was running on the virtual processor and if the process loaded in that virtual processor is in the ready state it could be assigned the virtual processor by the CALL INNER_TC('UNIDLE', VIR_PRO_ID). This has the effect of unloading the idle process and loading the process that was previously loaded. It should be noted that except for the special case of the idle process, switching processes is lengthy and, if done too frequently, could lead to thrashing problems.

The traffic controller can be invoked by the calls: 'STOP_PROCESS', 'CREATE_PROCESS', 'START_PROCESS', and 'DESTROY_PROCESS'.

'CREATE_PROCESS', PARAMETER_LIST is used to begin a new process. An entry for the process is made in the active process table.

'STOP_PROCESS' is used to put a process in the STOPPED STATE and the process is removed from the active process table and put in the stopped process table (SPT). The SPT is similar to the APT but it is referenced infrequently.

'START_PROCESS' is used to move a process from the stopped process table (STP) to the active process table and also from the stopped state to the ready state.

'DESTROY_PROCESS' is used to terminate the life

of a process. The process is removed from the APT or SPT and the memory manager process is signaled to disconnect the process from any connected segments.

d. Inner Traffic Controller (Kernel)

The inner traffic controller multiplexes the virtual processors with the physical processors[18]. There is a many to one correspondence from the virtual processors of the traffic controller to the physical processors. In addition there are the virtual processors assigned the system processes. The inner traffic controller uses the data base shown in figure 14. He is also responsible for the mapping registers (hardware segment descriptors) which contain the information shown in figure 15. Each physical processor has only specific virtual processors that can be multiplexed on it. Each virtual processor has a priority and a state (running, ready and wait). The inner traffic controller allows the virtual processor with the highest priority in the ready state to run on the processor. The wait pending bit[3,p.30] is used to avoid a race condition between the signal and wait primitives. The inner traffic controller is able to swap the virtual processors in and out of the processors by loading and unloading the appropriate execution state and mapping registers.

The inner traffic controller provides inner-process as well as intra-process services. He is invoked by a number of calls requesting information contained in the mapping registers or providing information

VIRTUAL PROCESSOR_#	STATE	PRIORITY	"COPY" MMU REG	SOFTWARE FAULTS

PROCESSOR MAP

FIGURE 14

ACCESS BIT	FAULT BIT	WRITTEN BIT	USED BIT	BASE ADDRESS	BOUND

MAPPING REGISTERS

FIGURE 15

to update the mapping registers. To supplement the hardware fault within the memory management registers the inner traffic controller maintains a set of software faults for each segment (segment fault, memory fault). This allows the inner traffic controller to interpret the hardware fault and generate an appropriate virtual fault.

INNER_TC('ASSIGN', SEGMENT_#, ACCESS) - a new segment number has been assigned with the indicated access. Load the appropriate register with the access, set the fault bit and the software memory fault.

INNER_TC('LOAD', SEGMENT_#, ABS_ADD, BOUND) - a segment has been loaded into memory, load the appropriate addresses in the mapping register and reset the memory software fault and fault bit if appropriate.

INNER_TC('UNLOAD', SEGMENT_#, WRITTEN) - the segment is being removed from memory, set the memory software fault and the fault bit and return the value of the written bit.

INNER_TC('WRITTEN_BITS', BITS) - an array reflecting the value of the written bits is returned.

INNER_TC('GET_USED_BITS', USED_BITS) - an array reflecting the value of the used bits is returned.

INNER_TC('SET_USED_BITS', USED_BITS) - an array is received reflecting the desired value of the used bits. The inner traffic controller sets the used bits to the desired values. The hypothesized hardware used bits are also set by hardware whenever a segment is referenced.

INNER_TC('LOAD_MMU', LOC_EX_STATE, ABS_ADD_MAP) - a request to load a virtual processor with a new process and create the memory management unit registers.

INNER_TC('UNLOAD_MMU', LOC_EX_STATE) - a request to unload a virtual processor and save the execution state in the indicated location.

INNER_TC('SET_SEG_FAULT', PROCESS_ID, SEGMENT_#) - a request to set the software segment fault in the data base (figure 14).

INNER_TC('IDLE') - a request to load the idle process and reduce the priority of the virtual processor to the lowest possible.

INNER_TC('PRE_EMPT_INT', VIRT_PRO_ID) - a request to generate a virtual pre_empt interupt to the indicated virtual processor. The inner traffic controller determines which physical processor the virtual processor is in and sends an appropriate hardware interrupt to that processor. If the virtual processor is in the wait state the interupt is held pending until the virtual processor is put in the ready state.

INNER_TC('UNIDLE', VIRT_PRO_ID) - a request to unload the idle process, reinstate the loaded process and restore the priority of the virtual processor.

The inner traffic controller is also invoked by the signal and wait. Signal and wait provide the synchronization between the system processes and the user processes. The inner traffic controller utilizes the signal

and wait primitives to change the state of the virtual processors and thereby control the multiplexing of the virtual processors to the real processors, based on their priorities.

5. Non-Distributed Kernel

The non-distributed kernel consists of the system processes. These processes have the characteristic that they function asynchronous to each user process. The system processes, as they are called, can reside in the local memory of each processor but their shared data bases will reside in global memory.

a. Memory Manager (System Process)

The memory manager process utilizes the Active Segment Table (figure 9) as a data base. The portion of the AST that contains system wide information will reside in global memory. The portion of the AST that only relates to a single processor can be distributed and will reside in local memory.

The memory manager process is responsible for two basic tasks: requests to bring segments into memory and requests to remove segments from memory. Other processes task him by use of the signal and wait primitives. The memory manager process has four tasks (entries): IN, OUT, LOAD, and UNLOAD. The IN and OUT are requests to load and remove a single segment. The LOAD and UNLOAD are requests to load and unload a number of segments.

The task to load a segment requires several considerations. Is the segment currently active (AST entry)? If it is, is it presently residing in global memory? If it is not in global memory does the access of the added process require that it be moved to global memory? How to alert the processes with copies? The AST provides all the necessary answers to render the proper decision as to where to load the segment.

At this time a better look at the AST is called for. It should be noted that every segment that presently resides in memory is active and its address can be determined from the AST. The virtual processor that it is in can also be determined as well as the segment number by which it is known within that virtual processor.

When a segment must be loaded into global memory (based on user access) there is a need to notify processors with a copy, of the segment, of the segments relocation. After the segment has been loaded in global memory, the memory manager process, tasked to load the segment, can determine form the AST in which processors the segment is presently loaded. These processors are sent SIGNAL('MEMORY_MANAGER', 'MOVE', UNIQUE_ID, ABS_ADD) where ABS_ADR is the global address of the segment. Each memory manager process that receives the signal('move') will check his local AST to determine which processes have the segment loaded and the segment number assigned and then CALL INNER_TC('CHANGE_ADD', PROCESS_ID, SEGMENT_#, ABS_ADD) for

each process that has the segment in local memory. The inner traffic controller will update the mapping register to reflect the new absolute address.

If a user requests access, and another user already has write access, there is a need to get the current copy moved to global memory. In this case the memory manager process attempting to load the segment must SIGNAL('MEMORY_MANAGER', 'MOVE_IT', UNIQUE_ID) and WAIT(PROCESS_ID, MSG). The processor with the current copy of the segment was determined from the AST. The memory manager process with the current copy, after receiving the signal('move_it'), will relocate the segment in global memory, CALL INNER_TC('CHANGE_ADD', PROCESS_ID, SEGMENT_#, ABS_ADD) and SIGNAL('MEMORY_MANAGER', 'MOVED', UNIQUE_ID, ABS_ADD). It should be noted that there is some synchronization required between the memory manager process and the inner traffic controller to insure the segment had not been written in during the time it took to move it and change the address.

As segments are loaded and unloaded the AST is updated appropriately. When a segment is removed from memory if it has been written in the segment is copied back to secondary storage.

The AST also provides a method of notifying processes of segment faults. If the memory manager process (for each processor connected with a loaded connected process) is notified when the access control list for a

segment is changed by SIGNAL('MEMORY_MANAGER', 'SET_SEG_FAULT', UNIQUE_ID) then every loaded connected process can be notified by CALL INNER_TC('SET_SEG_FAULT', PROCESS_ID, SEGMENT_#). For processes that are not loaded, the traffic controller is similarly called to set the software segment fault (figure 13). This means that the software segment fault will have to be set for connected processes when a segment is removed from the AST.

b. I/O Manager

The I/O manager is responsible for the external I/O. There could be more than one I/O manager process, conceivably one for each external device; corresponding kernel calls must be provided. For example there could be an I/O manager that handles all the external I/O to and from the user terminals. It is sufficent, at this point, to say that the I/O manager exists and handles external I/O.

6. Follow On Work

It should be re-emphasized that this is a design and not an implementation. Although the detail is left for further work, the design proposed forms a substantial basis upon which an implementation can be realized. The system process structure is provided for in the design; however, the system processes have been treated lightly and require additional work. The user interface (supervisor calls) presented is by no means an exhaustive list and could use further extension for additional supervisor services.

IV. CONCLUSION

The state of the art techniques and design methodology used to design secure operating system for multiple mini and maxi processors have been found applicable to the multiple microprocessor environment. The principal conclusion is that the operating system design in this thesis will make it possible to more effectively use modern microprocessors than has been possible in the past.

One question that is addressed concerns the operating system's ability to scale. Systems now available can support four or five microprocessors. Increasing that number of microprocessors quickly brings serious degradation because of the increased bus contention. The expected scaling factor is much better for this design. The bus contention has been significantly reduced - segmentation permits effectively using local memory instead of global memory.

This design supports a family of operating systems, not just one designed for a specific application. Sub-sets of this system can be constructed to provide the desired functions because the design used a loop free structure. Included family members range from a core resident tactical system to a virtual memory time sharing system.

Configuration independence is supported in this design. One or many physical processors can be added or subtracted from the system without affecting the workability of the

system. Similarly memory can be added or subtracted.

Security has been designed into this system. It was not added on as an afterthought. This design used a security kernel based upon a mathematical model to insure the security. A secure multilevel environment is provided by this system.

Commercial devices will soon be widely available to implement this operating system. The Zilog Z8000 series, microprocessor, for example will provide the segmentation and multiple domains necessary for an effective system. The present data buses are compatible and when used with this operating system allow a significant number of processors to be effectively used.

LIST OF REFERENCES

1. 'Architecture of a New Microprocessor', Computer, v.12 No 2, p.10, February 1979.

2. Mitre Corporation Report 2934, The Design and Specification Of A Security Kernel for the PDP-11/45, by W.L. Schiller, May 1975.

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

4. Lipner, S.B., "A Comment On The Confinement Property", Operating System Review, v.9, p.192-195, November 1975.

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

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

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

8. Janson, P.A., "Dynamic Linking And Environment Initialization In A Multi-Domain Process", Operating System Review, v.9 No. 5, p.43-50, November 1975.

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

10. LtCol Schell, R.R., "Computer Security: The Achilles' Heel of the Electronic Air Force?", Air University Review, v.XXX No.2, January 1979.

11. Millen, J.K., "Security Kernel Validation In Practice", Communications of the ACM, v.19, p.244-250, May 1976.

12. Denning, D.E., 'A Lattice Model Of Secure Information Flow", Communications of the ACM, v.19, p.236-242, May 1976.

13. Mitre Corporation Report ESD-TR-73-278, v.2, Secure Computer Systems: A Mathematical Model, by L.J. Lapadula and D.E. Bell, November 1973.

14. Mitre Corporation MTR-2932, A Software Validation Technique for Certification, Part I: The Methodology, by Bell, D.E. and Burke, E.L., November 1974.

15. Honeywell, Multics Processor Manual, p.6-1, Order Number AL39, Rev. 0, April 1976.

16. Madnick, S.E. and Donovan, J.J., Operating Systems, McGraw Hill, 1974.

17. Dijkstra, E.W., 'The Humble Programmer', Communications of the ACM, v.15, p.859-866, October 1972.

18. Reed, D.P., Processor Multiplexing In A Layered Operating System, Master's Thesis, Massachusetts Institute of Technology, MIT/LCS/TR-164, 1976.

19. Janson, P.A., Using Type Extension To Organize Virtual Memory Mechanisms, Ph.D. Thesis, Massachusetts Institute of Technology, MIT/LCS/TR-167, 1976.

SECURITY KERNEL DESIGN FOR A MICROPROCESSOR-BASED, MULTILEVEL, ARCHIVAL STORAGE SYSTEM

by

Aaron Ray Coleman
Captain, United States Army
BAM, Auburn University, 1972

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1979

Dean of Information and Policy Sciences

ABSTRACT

This thesis is a detailed design of a security kernel for an archival file storage system. Microprocessor technology is used to address a major part of the problem of information security in a distributed computer system. Utilizing multiprogramming techniques for processor efficiency, segmentation for controlled sharing, and a loop-free structure for avoiding intermodule dependencies, the Archival Storage System is designed for implementation on the Zilog Z8001 microprocessor with a memory management unit. The concepts of a process structure and a distributed kernel are used in providing management of the shared hardware resources of the system. The security kernel primitives create a virtual machine environment and provide information security in accordance with a non-discretionary security policy.

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.

There are several persons who have aided me greatly in the preparation of this thesis whom I expressly want to thank. My thesis advisor, Lt. Col. Roger Schell, tutored me in many long sessions and used many hours of his time reading my drafts. My mother-in-law, Iva Jewel Tucker, edited and styled every word I wrote and forced me to consider the exact meaning of each word.

Finally, and most importantly, I want to thank my wife, JoAnn. She assisted me in more ways than I can enumerate and always provided encouragement when all seemed impossible.

I. INTRODUCTION

This detailed design of a security kernel provides a basis for implementation of an archival file storage operating system. The system is intended to store files for an array of computer hosts at multiple information security levels. The design presents algorithms and data structures which can be implemented on microprocessor hardware available today, to provide economical and secure storage. Controlled sharing of information and multilevel security were the key design goals. Multiprogramming is the technique used to improve efficiency of the system which is primarily performing input and output operations. A loop-free structure is used to avoid undesirable dependency loops [1]. This allows modules to be changed without introducing changes in other modules.

There are two components of the Archival Storage System: 1) the Supervisor and 2) the Security Kernel [2]. The Supervisor (the subject of separate research [3]) supports all user services: 1) hierarchical file system, 2) discretionary access controls, and 3) protocols for communication. The Supervisor operates outside the Kernel domain on a virtual machine created by the Kernel primitives. The Supervisor's privilege-restricted domain has access only to a subset of the machine instructions, thus needing the Kernel primitives to accomplish tasks such as input or output.

The Security Kernel described in this thesis manages

the real resources of the hardware system: 1) memory, 2) microprocessor, 3) external devices, and 4) input/output ports. It is also responsible for mediating all non-discretionary access to information. The Kernel operates in the most privileged domain of the machine and therefore has access to all machine instructions.

A. BACKGROUND

Microprocessors have become affordable, prolific, and powerful computing resources. The result of these attributes is the use of microprocessors in applications previously requiring much larger and more expensive processors. Additionally, new applications which can now be economically computerized are being seriously explored.

Conversely, software has become more costly. Microprocessor operating systems and applications programs continue to be highly specialized, thus failing to reasonably exploit the potential of the microprocessor. The specialization of software for microprocessors also perpetuates problems such as I/O format incompatibilities which occur when information exchange among processors is desired.

Information security on microprocessors has been completely ignored to date, or handled with ad-hoc attempts at a solution. However, this issue is becoming increasingly important as the uses of microprocessors continue to be expanded. For example, the Department of the Navy is investigating the use of microprocessors on

small ships for automating shipboard administrative functions [4]. Information security for such functions is a major requirement which cannot presently be met.

Proposing a solution to the above problems, a high-level design for a secure operating system for microprocessor-based systems has been outlined by O'Connell and Richardson [5]. The design goals of that operating system were configuration independence, distributed processing, multiple protection domains, multiprocessing, and multiprogramming. Because such a broad, general operating system is not always required, the design provided for a family of operating systems. A family member could use a subset of functions for a specific application while allowing later extensions. This thesis presents the detailed design for such a family member.

B. BASIC CONCEPTS

The Archival Storage System can be the nucleus of a secure, distributed multiprocessor system. It provides 'data warehouse' facilities for multiple host computers in the network. A host may be operating at a single security level, or simultaneously at several security levels without affecting the Archival Storage System. Information storage with multilevel security is provided for each host connected to a port of the warehouse. Additionally, the data warehouse is the mechanism for providing controlled sharing among the hosts. Thus, we can apply microprocessor

technology to address a significant part of the larger multilevel security problem [6] for distributed systems.

A subset of the O'Connell and Richardson design has been selected as the basis for the detailed design of the Archival Storage System. (The subset chosen omits the provisions for multiprocessors, dynamic linking, demand segmentation, "transient" processes, and a user domain.) The Supervisor, protocols, and interfaces to the host computers are presented in a parallel thesis by Parks [3] while detailed design of the Security Kernel is presented in this thesis.

There are two components of the Archival Storage System Security Kernel which reside in the privileged domain of the machine: 1)the distributed kernel and 2) the kernel processes. From a logical view, some kernel procedures are distributed among all the Supervisor processes in the system, with the remaining procedures forming kernel processes. These kernel processes perform functions that are asynchronous to the supervisor processes and are responsible for the shared resources of the system (processes, processor, memory, input/output).

1. Definition of a Process

A sequential process can be conceptualized as an execution point and an address space which is a logical rather than physical entity. All procedures that are in the flow (or locus) of control are in the address space. In a distributed operating system, the locus of execution

includes those operating system functions which are logically part of the user process. The distributed operating system is divided into procedures which are called in normal fashion, but are located in the privileged domain.

2. Multiple Protection Domains

One requirement for design of a security kernel is isolation of the kernel procedures to make them tamperproof. A way this can be achieved is to arrange the process address space into hardware or software protection domains. Domains need not be hierarchical, but in this case they are. Hierarchical domains are commonly called protection rings [7].

Each level in the hierarchy is more privileged than the preceding level. In the Archival Storage System only two domains are necessary. Other levels must be added to protect the Supervisor if the design is extended to include user applications. The distributed Kernel resides in the most privileged domain and may access any segment within the address space of a process. All systemwide databases are in the kernel domain. Violation of the confinement principle described by Lampson [8] and Lipner [9] would occur if such information could be passed to other domains.

The Supervisor operates in the outer or least privileged protection domain where access to segments and external devices is restricted. Only those databases which

are "process local" may be accessed. This does not prevent sharing since different segment numbers and access rights for each process can be interpreted and enforced by the kernel. Each Supervisor process is required to remain at a specified security level within its domain.

Protection domains may be created by either hardware or software. Software implementations of protection domains (as in the early Multics [10]) are feasible, but result in a degradation of efficiency. This performance loss is unacceptable in many applications. In large processors a hardware ring mechanism is sometimes used to provide the implementation [7]. This general ring mechanism is not available in current microprocessors, but two domain machines are available. When supplemented by ring-crossing software, this will provide the desired multiple domains.

3. Segmentation

A segment is defined as a logical grouping of information [11], while segmentation is a technique for managing segments within an address space. A process's address space consists of a collection of procedures and data segments. All address specifications require the segment specification and the offset within the segment (i.e., a two-dimensional address). Segments are therefore distinctly visible to the user. Unlike pages, segments are arbitrarily sized and logical units with logical attributes to describe them.

Attributes of segments are contained in a structure called a segment descriptor. The descriptor associates segments with address in memory, size, and access allowed. Maintaining all of the descriptors of the segments of a process in a descriptor list allows the address space of the process to be easily managed.

Segmentation offers benefits as a memory management scheme. The key advantage is the ability of multiple processes to share segments without the requirement of maintaining multiple copies in memory. Other favorable characteristics of segmentation are control of memory waste due to fragmentation, creation of user virtual memory, dynamic linking of modules, and enforcement of controlled segment access.

Segmentation eliminates the need to duplicate a segment when shared. Having only one copy saves memory and eliminates the problem of conflicting data which occurs when multiple copies are maintained. Even more central to segmentation is the ability of cooperating processes to communicate with each other through shared segments. Inter-process synchronization and communication are necessary functions in a multiprogramming environment.

4. Information Security

Most users of computer systems are required to safeguard information from unauthorized access. Examples abound: government (classified information), corporations (trade secrets), banking (electronic funds transfer), and

all users of personal data (privacy act). This requirement is not relaxed when microprocessors are used instead of (or in support of) large computer systems. Dedicating a device to a specific security level (dedicated mode) [6] is a method commonly used to meet the security requirement. This solution is unsatisfactory for any user with a requirement to utilize data at more than one access class.

Another solution to the problem of accessing information at different security levels is to operate in the multilevel mode. In this case both users and information at different security classes exist simultaneously on the same computer system. Users are not permitted to access information unless authorized by the security policy in effect.

In the dedicated mode all security measures are external to the computer system (e.g., perimeter fencing, guards, door locks, etc.). When a multilevel mode environment is used, controls must be internal as well as external. Attempts at internal controls have been tried by adding security measures to existing systems with unsatisfactory results. Numerous cases are documented of penetrations (i.e., unauthorized access) of these systems [6]. Intuition rather than sound design was the methodology used in these unsuccessful attempts at security.

Internal controls must be designed into a system

from its conception. The approach to designing these controls is the security kernel methodology. The first step using the security kernel methodology is to define the security requirements. From this definition a conceptual design is created. The conceptual design is actually a mathematical model which can be rigorously proven and provides the basis for testing (certifying) [2] all subsequent implementations.

Three things are requisite before a system can be secure using the security kernel concept: 1) The kernel must be isolated or tamperproof. Obviously if a penetrator can change the kernel software, then the behavior of the kernel can be modified. 2) The kernel must be invoked on every attempt to access information. This requirement can be met by initial software interpretation of access on the first call to a segment. Thereafter, hardware can enforce the access criteria. 3) The kernel must be subject to certification. Proof of the mathematical model must be followed by thorough testing of the implementation to insure that each input yields the desired output. Since hardware and software are involved, both must be tested before the kernel can be certified.

As previously stated, the first step in the design of the secure computer system is to define the security requirements. A properly designed computer system is then secure with respect to that definition or policy. A security policy consists of the external laws, rules, and

regulations that establish what access is to be permitted. Two distinct types of security policy exist: 1) non-discretionary and 2) discretionary.

Non-discretionary policy involves comparing the requested (i.e., the information object's) access class (oac) with the access class of the requestor (i.e., the subject) (sac) to insure that they are compatible. For example, in the Department of Defense security policy a secret cleared individual (subject) may have access to documents (objects) which are classified as secret, confidential, or unclassified.

The relationships between different access classes can be represented by a lattice structure [12]. This lattice structure is totally ordered if all classes are related. When the classes are either related or disjoint the lattice is partially ordered. The lattice structure interprets the authorized access based on the relationships between two labels. The lattice structure abstraction is important because it seems to represent most practical security policies. By changing the interpretation of labels in the non-discretionary security module, a different policy can be implemented so that, for example, Privacy Act requirements are as enforceable as Department of Defense security policies.

The following interpretation defines the access permitted in a computer system (where "%" is defined to mean unrelated) in terms of subject access class (sac) and

object access class (oac):

sac = oac, read/write permitted

sac > oac, read permitted (read down)

sac < oac, write permitted (write up)

sac % oac, no access

DOD security policy is represented by a partially ordered lattice since security classifications are composed of a classification level and a category (e.g., secret, cryptographic or confidential, nuclear).

Discretionary controls provide a refinement of the non-discretionary access. A common example of discretionary controls involves checking an access control list before allowing an access. This allows authorized subjects (users) to specify who may use that segment within the confines of the non-discretionary policy. The DOD "need-to-know" rule is an example of discretionary policy. In the Archival Storage System non-discretionary policy is enforced by the Supervisor, based on both the host and the user.

C. STRUCTURE OF THE THESIS

This thesis presents the detailed design of a portion of the security kernel for an archival file storage facility (distributed kernel procedures and memory manager process). Levels of abstraction are used to reduce the complexity of the hierarchical Archival Storage System. Level 2 contains the Supervisor and operates in the virtual environment created by the Kernel. The Supervisor

does not control the hardware of the system but applies the hardware resources only by appealing to the functions of the Kernel. Calls to procedures at different levels may only be made in a downward direction and corresponding returns only in an upward direction (i.e., the Kernel may not call upon the Supervisor to accomplish any task). This restriction, rigidly enforced at all levels of abstraction in the design, reduces the number and type of interactions of the system.

Figure 1 shows the process structure of the system with the distributed and non-distributed kernel. The asynchronous Memory Manager and I/O Manager are kernel processes. The remaining kernel procedures are distributed in all the supervisor processes.

In the next chapter the details of the design are presented. Although this is not an implementation, the Zilog Z8001 Microprocessor [13] with the Z8010 MMU Memory Management Unit is used as the hardware base for this research. Choices made during the design process were often influenced by the hardware features available. The Z8000 family of devices is not mandatory for implementing the Archival Storage System. Other microprocessors exist which are capable of supporting a secure system.

Algorithms and data structures are presented in the high level language PLZ/SYS [14]. PLZ/SYS is a Pascal-like, block-structured language. Designed by Zilog, the language can be compiled to Z8000 instruction code.

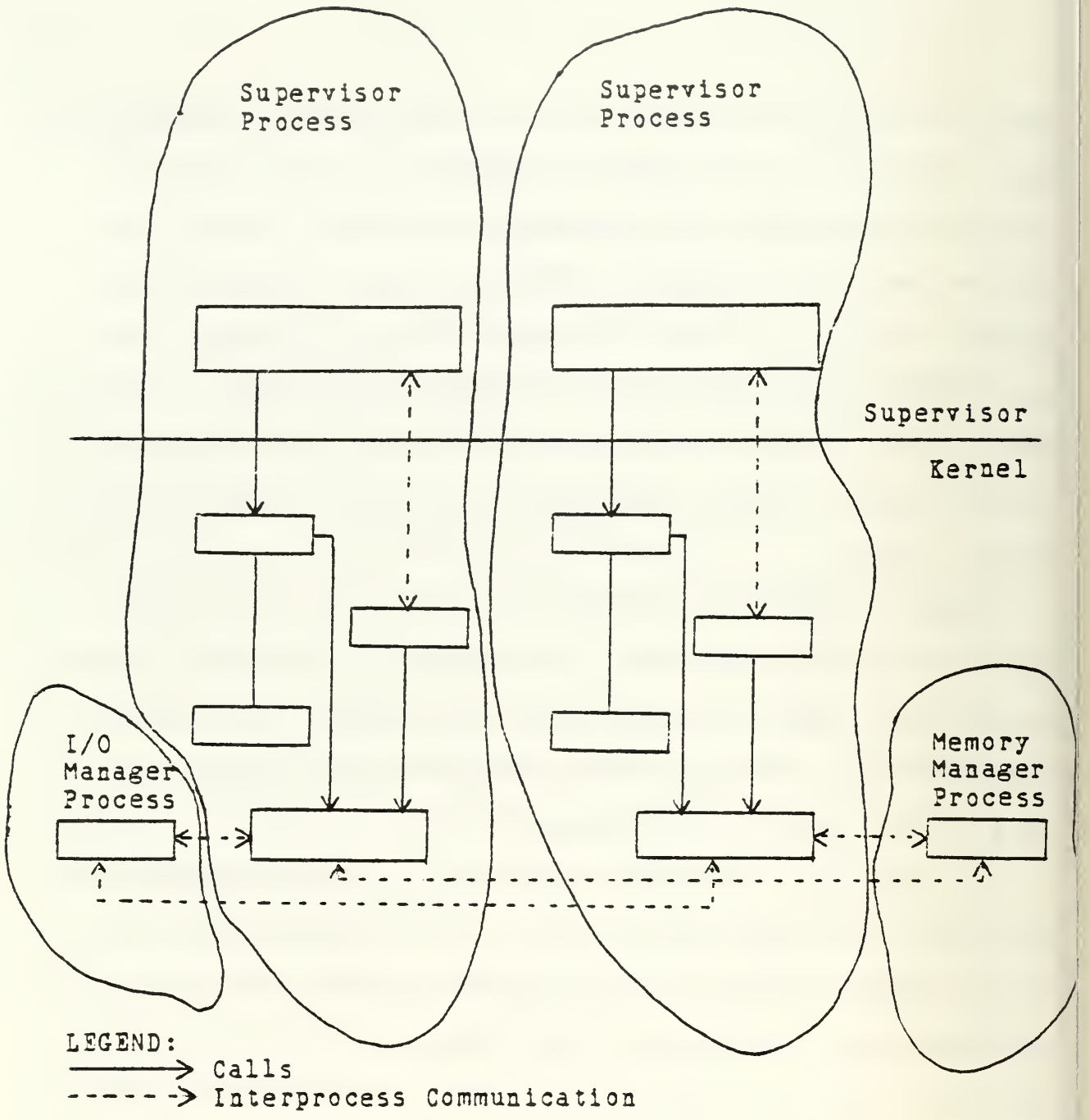


Figure 1. Process View

PLZ/ASM [15] is used where machine level instructions for direct hardware manipulation are required. These two languages for the Z8000 are used because of the capability of linking modules of either language to the other. Additionally, PLZ/ASM has the same high level data structures and structured control mechanisms present in PLZ/SYS.

The conclusions reached during this research are presented in the last chapter. Topics for further research and implementation are identified, including those in the area of secure systems. With this multilevel data-store, a secure distributed microprocessor system can be implemented using communications lines to interface distributed processors of the network to the "data warehouse."

II. DETAILED DESIGN

A. HARDWARE REQUIREMENTS

Theoretically any processor hardware is usable for secure computer systems. However, in practice certain hardware features are essential for efficiency. In addition, complexity of the software portion of the security kernel is highly dependent upon the capabilities of the hardware. Because simplification of the kernel results in an easier proof of correctness, it is worthwhile to use additional hardware to achieve security.

One essential hardware feature is a segmentation mechanism. Segmentation allows the use of one uniform type of information object, the segment. Kernel software which deals with logical objects is then simplified. Paging hardware without segmentation does not provide a correct structure for objects, since pages are physical objects with physical attributes. For example, an access right is a logical attribute. Applying an access right to a physical object (as is the case for IBM 370 protection "locks" [11]) obscures the purpose of the attribute, is out of context, and adds complexity to the supporting mechanism.

A segment address consists of a segment name and an offset within the segment. This logical address must be transformed into an absolute address before it can be used. While software is capable of making this transformation, hardware can perform the mapping more

efficiently and simultaneously check access while doing the mapping.

Multiple execution domains are also considered essential in hardware. This feature is used in current computer systems to protect the operating system from applications programs, but until recently this feature has not been available in microprocessor hardware. In the initial Archival Storage System implementation only two domains are necessary since applications programs do not exist inside the boundaries of the system. Protecting the Kernel from the Supervisor is the only domain protection required. If user utilities are added to the design, then another domain will be necessary to protect the Supervisor from user tampering.

With the introduction of Zilog's Z8000, the above hardware features are available in the microprocessor category. The design of the Archival Storage System is targeted toward a hardware system based upon the Z8001 segmented microprocessor [16] and the Z8010 MMU Memory Management Unit [17]. The Z8001 is a 16-bit two-domain microprocessor which produces a 23-bit segmented address. The Z8010 MMU maps the 23-bit logical address into a 24-bit absolute address and allows the capability of addressing up to 128 segments of 64K bytes each in the two-dimensional memory space.

In addition to the address mapping hardware, the MMU also provides memory access protection. Segment access may

be set to write (read implied), read, or execute only. When an unauthorized access is attempted, the MMU prevents the access, then sends a trap (or fault) signal to the microprocessor. A trap is an internal interrupt which is synchronous rather than asynchronous to the cycling of the processor and must be resolved by the processor before processing can continue.

The microprocessor also supports two protection domains. The MMU provides the implementation of two hardware rings by checking for system or user status on each access to a segment. The bit in the MMU which specifies system or normal mode, thus specifies which segments are accessible in just the Kernel ring and which segments are also accessible in the Supervisor ring. Thus a process must cross into the Kernel ring to access the Kernel primitives. If more than two rings are required, an additional MMU (up to eight total) may be employed per ring.

The hardware also supports resource control by limiting the use of certain machine instructions. In the system mode all machine instructions can be executed. When in user mode, the hardware will not allow the use of input/output instructions, certain machine control instructions, or special input/output instructions (used to load and control the MMU). Thus the Kernel can control the microprocessor, the main memory (through the MMU), and all external devices.

Hardware features other than those described above are indicated from a performance standpoint. For instance, a direct memory access (DMA) device could make memory to memory, memory to port, or port to memory transfers faster than similar transfers under direct CPU control. This would allow the CPU to continue with other tasks while the DMA is processing the data transfers. Protection of memory can still be realized by routing the DMA through the MMU. The DMA would have to be "smart" enough to handle an access violation trap or the Kernel would have to guarantee, by MMU set-up, that the DMA would not violate the security policy. This type of hardware is not crucial to the design at this level, and the decision on its use is left to the implementor.

The MMU does lack a descriptor base register capability [10]. Process switches without this facility require at least selective unloading and loading of the descriptor registers in the MMU, and a process switch would take roughly two (2) milliseconds to accomplish in this manner. It is evident that process switching may lead to thrashing problems if done too often. There are ways the implementor might avoid this problem (e.g., dedicating an MMU to each process, then switching MMUs rather than loading/unloading a single MMU).

B. PROPOSED KERNEL DESIGN

1. Notation

Notation is important in making algorithms

understandable. It should not, however, require more thought to understand the notation than the central concept. Since this thesis presents a detailed design, a notation as close as possible to an actual language which can compile to Z8000 machine code was desired. PLZ languages are used as a notation to illustrate the data structures and procedures. However, the code as shown in the figures cannot be directly implemented. Among other changes, procedure order has been rearranged to make explanation of the modules more logical. This change would violate a PLZ/SYS implementation rule that procedures must be declared before they can be invoked.

The details of the actual PLZ/SYS language implementation may be different from that assumed in this thesis. In particular, the specific method of parameter passing between PLZ/ASM and PLZ/SYS is unknown at this time. The implementor should carefully investigate how passing of parameters in the actual language implementation affects the interfaces between modules.

Because of the terminology used in the Z8000 Microprocessor specifications, the Supervisor may be referred to as operating in the normal or user mode. If the term system mode is used, it refers to the Kernel domain of execution.

2. Kernel Overview

The distributed Kernel modules exist on three levels (figure 2). Each module creates a different level

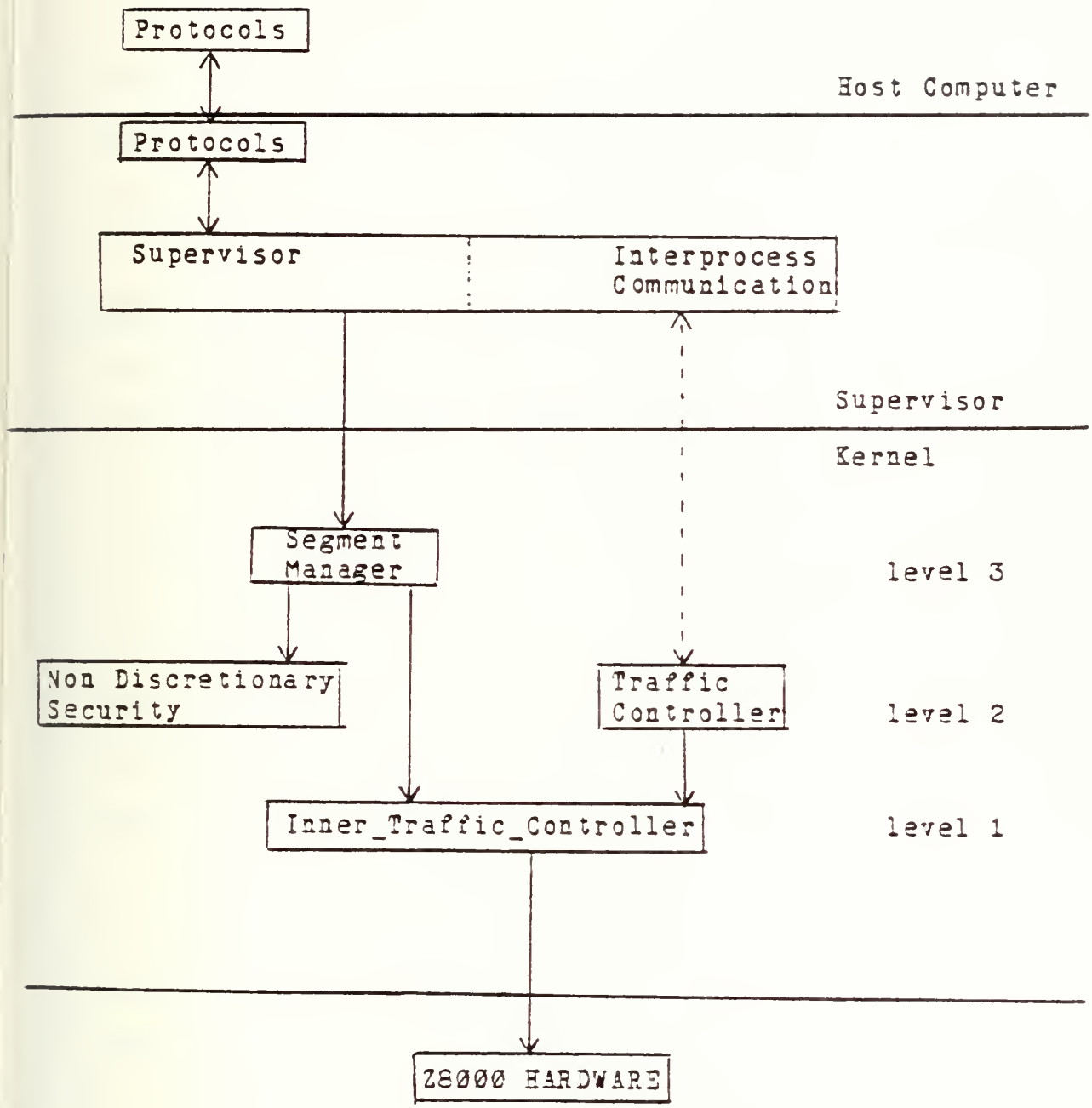


Figure 2. Hierarchical View

of abstraction [18]. At level 1 or the innermost level is the Inner_Traffic_Controller. Its primary task is the control of virtual processors and the multiplexing of virtual processors onto the real processor. The Inner_Traffic_Controller uses the Virtual Processor Table as a management tool for this multiplexing of virtual processors.

At level 2 is the Traffic_Controller. The Traffic_Controller creates the sequential process abstraction [17]. A process can be in one of two states: 1) blocked or 2) unblocked. When blocked, it must wait for the occurrence of some event. Since the process cannot proceed until that event occurs, the virtual processor is freed and then allocated to another process. When unblocked a process is either ready or running. In the ready state, the process can run when a virtual processor is assigned to it. The ready state can be entered from either the running or blocked state (figure 3).

The Non_Discretionary_Security Module is also on level 2. This module is charged with interpretation of the security policy in effect. It compares the two labels which are passed to it and determines the relationship of the labels based on a lattice structure known to the module. This relationship is then used by the kernel to determine authorized access to objects (segments or parts). It is emphasized that the Kernel makes decisions about access based on relationships (=, <, >, not related)

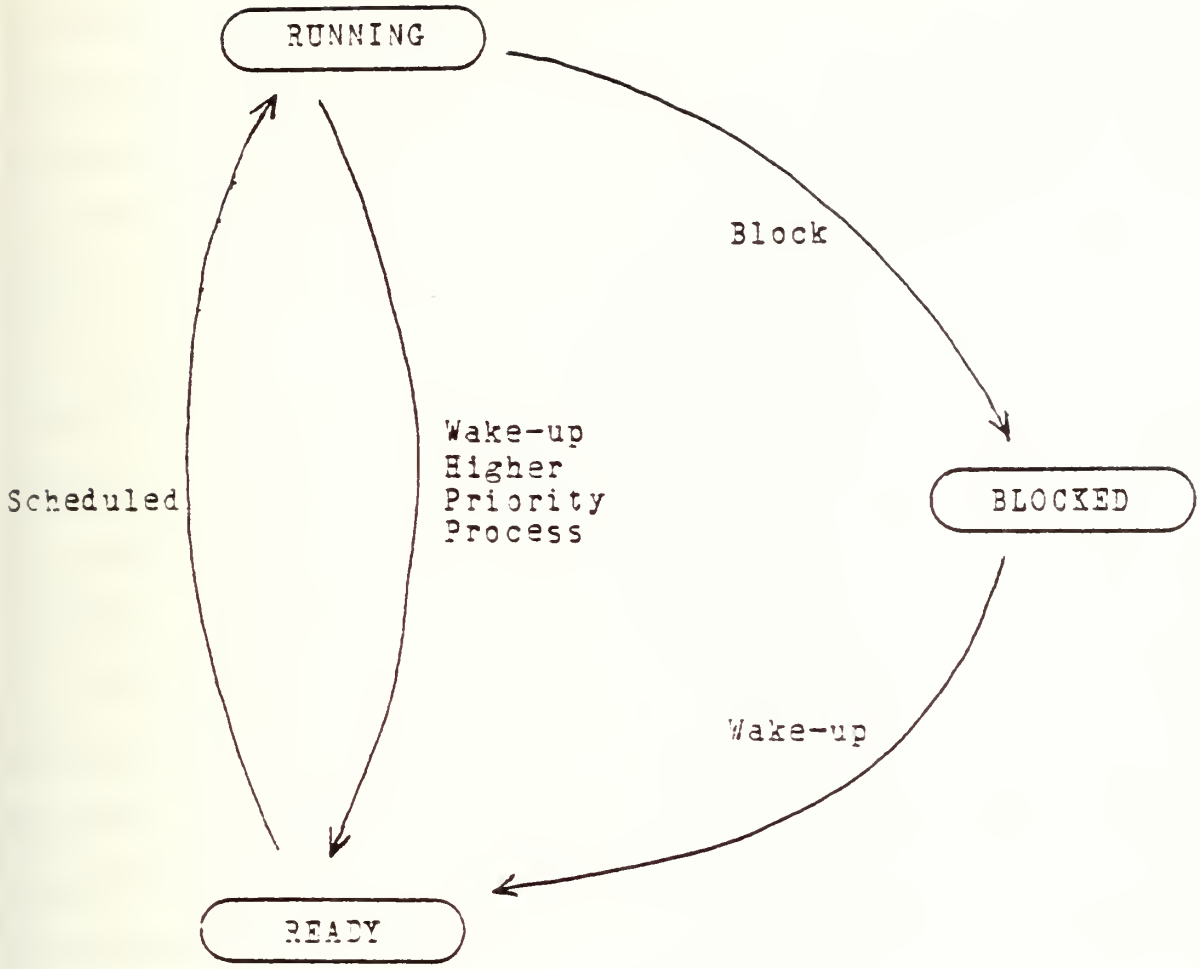


Figure 3. Process States

and not on the labels themselves. The Non_Discretionary_Security Module is the only module in the Kernel which makes any interpretation of security labels. This allows most of the practical security policies to be implemented simply by changing the Non_Discretionary_Security Module.

At level 3 is the Segment_Manager. Using the MMU mapping to real memory provided by the hardware, the Segment_Manager creates a segmented virtual memory for the process. Because of the limitations of the hardware (lack of a paging mechanism), segments are not dynamically allocated real memory. The size of a requested segment is fixed (or determined) at the time it is created and may not change. The Supervisor has several options in order to handle the problem of growing segment size: 1) Allocate the maximum size to every segment which is wasteful of memory, 2) copy the segment into a larger segment whenever the size changes which is wasteful of processor cycles, 3) create a "super-segment" as a collection of segments, or 4) some combination of the above. By requiring the Supervisor to handle this problem, the initial Kernel implementation is simpler.

The whole segment must be swapped into main memory in order to be used. The MMU supports segments ranging in size from 256 bytes to 64K bytes in multiples of 256 bytes. Additionally, the hardware forces another constraint on the design. Without paging, two allocation

schemes are available to the designer: 1) a demand segmentation memory management scheme (load the segment in response to a fault) or 2) a partitioned allocation scheme. In this design a partitioned allocation scheme is used to make the Kernel less complex. Part of the burden of memory management is then forced on the Supervisor. The Supervisor of each process is given a fixed amount of linear "virtual core". Linear 'virtual core" is distinguished from the two-dimensional virtual memory created by the segmentation. The Supervisor, by requests to the Kernel, may fill virtual core with segments as it chooses. The Supervisor of each process must manage its own virtual core and fit any segments it uses within the boundaries of this virtual core. The partitioned allocation portion of the memory management scheme is supported by the Memory_Manager process of the non-distributed Kernel.

The non-distributed portion of the Kernel resides in two kernel processes: 1) Memory_Manager and 2) I/O_Manager. These two processes are responsible for actions which are not logically part of the supervisor processes because they can function asynchronously to the processes. The Memory_Manager moves segments within the physical memory space of the system. These transfers may be main memory to main memory, main memory to secondary storage, or secondary storage to main memory. Main memory to main memory moves are made because of a design decision

to restrict sharing of the same copy of a segment unless at least one of the sharing processes has write permission to the segment. Whenever two processes share a segment and neither has write access, two copies of the segment will exist--one in each virtual processor local memory. This trade-off results in less complexity in the kernel and when the design is expanded to a multiprocessor implementation, bus contention is minimized [5]. The problems associated with the existence of multiple copies in memory are not present since the segment is not writeable.

Whenever a segment is to be shared and is writeable, then the segment must be moved to the real processor global memory. Movement of the segment is easily accomplished by updating the appropriate MMUs to reflect the new location of the segment. This concept of a process local and global memory is analogous to processor local and global memories in multiprocessor systems. In those systems, each real processor owns a local memory, while the system controls the global memory used by all processors for shared information.

The I/O_Manager is responsible for routing segments across the system boundary, viz., moving data between external ports of the system and main memory. The I/O_Manager does not try to interpret the data, but simply provides a transfer service. All the ports have specific security classifications and are hard-wired. This allows

the I/O_Manager to function without requiring labels or other security mechanisms to determine access class. Having all Hosts at a fixed security level is a design choice for the Archival Storage System. Hosts can be at multiple levels if the design is modified to accept "trusted" labels. In the present design the Host computer is required to be at the level of the port and to handle data consistent with the security policy in effect.

Since the hardware does not completely support the ring structure, software (Gate_Keeper) is needed for the ring-crossing mechanism and thus isolation of the Kernel. All calls to the distributed Kernel and interprocess communication with the non-distributed Kernel from the Supervisor must pass through the Gate_Keeper. The function of the Gate_Keeper is to provide the sole entry point or gate into the Kernel ring, validate the call and arguments, and transfer the call to the appropriate kernel module. If a call is made incorrectly the Gate_Keeper sets a return message to an error code, and returns without further action. The Gate_Keeper is the ring-crossing mechanism of the Archival Storage System.

3. Gate Keeper Module

The Gate_Keeper Module (shown in Appendix A rather than as a figure because of its length) consists of procedures and primary data structures and is the sole entry point into the Kernel from the Supervisor. The Gate_Keeper Module is written in PLZ/ASM since it is a

trap handler. (The user registers must be saved when the handler is invoked which requires access to the hardware.) When the Supervisor wishes to invoke the Kernel it must put the argument list and space for any return message in a segment with read/write access in the Supervisor ring. When the system call is made, the pointer to the arguments is required to be in a double register. The system call instruction is then executed, with the function-code for the requested Kernel procedure as a parameter within the instruction. This causes the machine to save the program counter, flags and control word, and the instruction itself on the system (kernel) stack. An unconditional jump (hardware initiated) is then made to the Program Status Area (a vector table) (figure 4) where the machine state for the system call instruction is fetched. The Program Status Area is established at system generation and consists of "frames" which contain the machine state and location of the interrupt and trap handlers. The processor then begins execution in the Kernel ring.

The Gate_Keeper first saves the user processor registers and retrieves the pointer to the argument list. If the argument list is located in a read/write segment of the calling (Supervisor) ring, a copy of the argument list is put onto the system stack. However, if the area indicated by the calling ring is not in the read/write address space of the process, the Gate_Keeper will not return an error code. (There is no place to return it!)

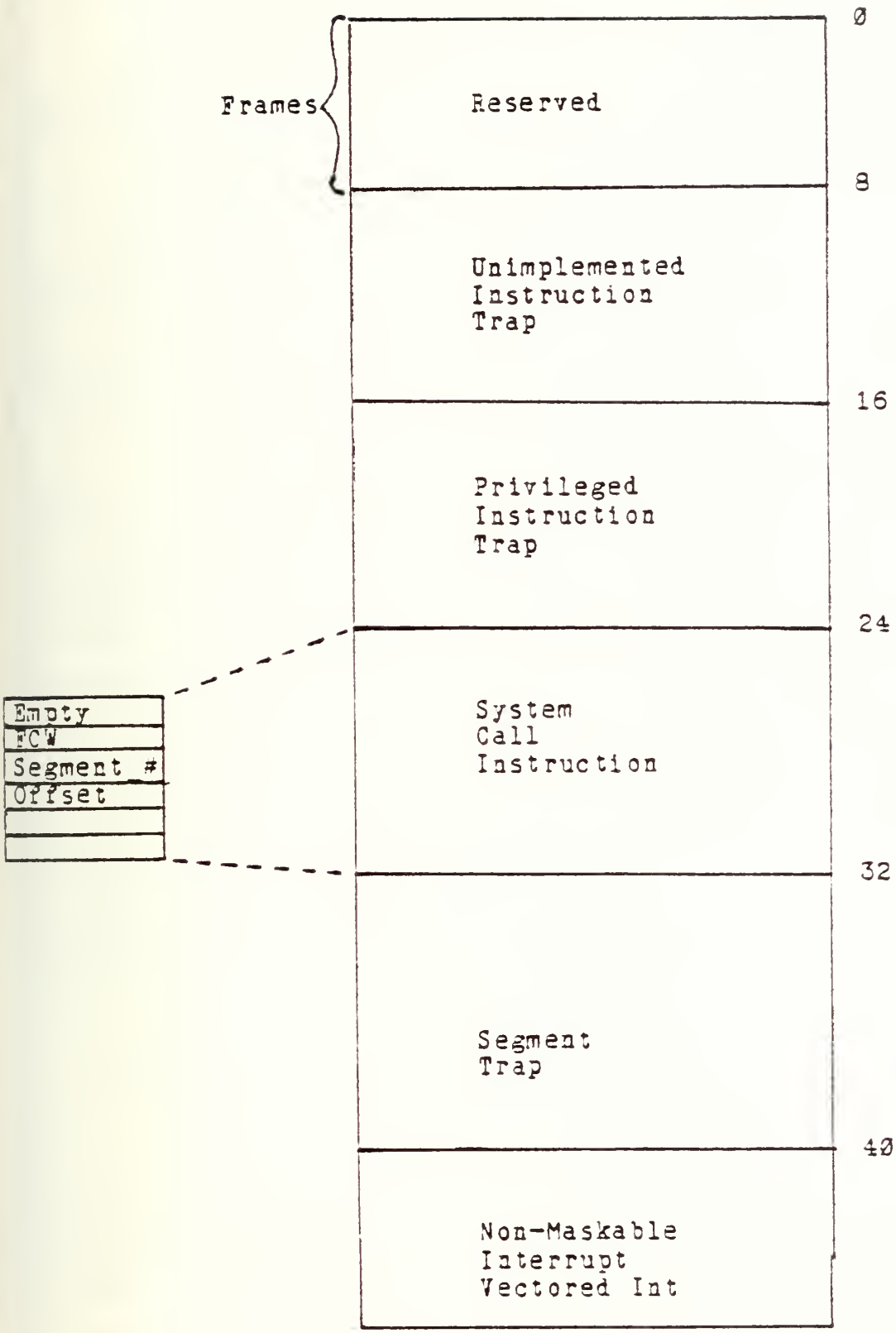


Figure 4. Program Status Area

The Gate_Keeper restores the user environment and makes a normal return.

The Gate_Keeper uses a table (figure 5) to check the range of the function code. If the Gate_Keeper now discovers an error during the validation process, it sets the return message to an error code, copies the argument list back to the user area, and returns in the usual way. If the call is valid, the Gate_Keeper calls the appropriate module (e.g., Segment_Manager) at the requested entry point into the module.

When the module has completed the requested task it returns to the Gate_Keeper. The return message is then copied to the user's return argument, and a return to the user ring occurs. All entries into and exits from the Kernel are through the Gate_Keeper.

Parameter passing to and from the Kernel is by value only. Since implementation details of how PLZ modules pass parameters are unknown, the decision on the precise mechanism for argument passing is left for the implementor. It may be best to align the method of parameter passing as closely as possible to the method used by the PLZ/SYS language.

4. Segment Manager Module

The Segment_Manager is responsible for managing the segmented physical memory and uses the Known_Segment_Table (KST) as its primary database. In keeping with the loop-free structure and since the

Function_Code

Function	Number of Parameters	Para-1 Length	Para-2 Length	...	Return Para Length
Create_ Segment					
Delete_ Segment					
Make_ Known					
Terminate					
Swap_In					
Swap_Out					
Block					
Wake_Up					

Figure 5. Parameter Table

Segment_Manager is the only module at level 3 of the Kernel, only calls external to the Kernel domain may be made to the Segment_Manager. There are six entries into the Segment_Manager in this implementation:

1) Create_Segment
2) Delete_Segment
3) Make_Known
4) Terminate
5) Swap_In
6) Swap_Out

a. Known Segment Table

The data structure used by the Segment_Manager to manage segments is the Known Segment Table (KST). The KST is a "process local" data structure and contains an entry for each segment which the process has declared an intention to use (viz., "made-known"). The segments may or may not be located in main memory. If a segment has an entry in the KST, then the segment is described as known to the process. In this design it will also have an entry in the Active Segment Table (AST--a Memory_Manager database explained later) and can be described as active. The KST (figure 6) is indexed by the segment numbers (Segment_#) which are assigned by the Segment_Manager. The Segment_# also corresponds to the MMU descriptor register for the segment. The ASTE_# is the Active Segment Table entry number and is obtained from the Memory_Manager. The ASTE_# is the "handle" which is passed to the

Memory_Manager when necessary to identify a particular active segment. The Size field is an integer which is the size of the segment in bytes divided by 256. All segments are created in multiples of 256 bytes because of MMU constraints. An upper bound (Max_Segment_Size) is placed on the segment Size by the design (explained later). A flag known as In_Core is used to indicate whether the segment is in main memory or on secondary storage.

The last field in the KST entry is the access class of the segment. This is a label which indicates the security classification of the segment. Interpretation of the Class to determine an access mode (read or read/write) is performed by the software (by a call to Non_Discretionary_Security) on first reference; thereafter the access mode is enforced by the MMU.

Figure 6 shows both the logical view and the PLZ variable declaration for the KST. Max_KST_Size is hardware dependent and is equal to the maximum number of segments which can be mapped by the MMU. To access an element of the database the following notation is used:

KST [Segment_#]. ASTE_#

If Segment_# is equal to 103 then the above statement will reference the ASTE_# field of the KST entry for segment number 103.

b. Creation and Deletion of Segments

Create_Segment and Delete_Segment are two of the six Supervisor entries into the Segment_Manager.

Segment_#

ASTE_#	Size	Access Mode	In Core	Class

Known Segment Table Logical View

```
Type
  KST_Entry Record [ASTE_# AST Index
                    Size Integer
                    Access_Mode Integer
                    In_Core Byte
                    Class Longword]

Internal    ! Internal to the Segment_Manager !
     KST Array [Max_KST_Size KST_Entry]
```

Known Segment Table Database Definition

Figure 6. Known Segment Table

Create_Segment (figure 7) is the function which adds a new segment to the Archival Storage System after validating the parameters which are passed. The creation of a segment is accomplished by requesting the Memory_Manager process to make an entry in the Alias Table and to allocate storage on secondary media.

The Alias Table is a database which is maintained by the Memory_Manager. It is a result of the aliasing scheme used by the Kernel to prevent passing systemwide information (such as the unique identification of a segment) out of the Kernel [20]. The alias of a segment is the segment number of a "mentor" segment (a process local variable) and the entry number in the Alias_Table. The principal implication of the aliasing scheme is that a mentor segment must be known before a segment can be created. The Alias Table will be further explained in a succeeding section.

The arguments which must be passed to Create_Segment are the Mentor_Segment_#, the desired Entry_# (in the Alias_Table), the Class of the segment (a label), and the desired Size of the segment. The KST is searched to insure that the Mentor segment is known. Next, Non_Discretionary_Security must be called to determine if the segment is compatible [2]. (To be compatible, a mentor segment classification must be less than or equal to the created segment.) The compatibility check can be performed in the Segment_Manager or the Memory_Manager. In addition

```
Create_Segment Procedure (Mentor_Segment_# Integer
                          Entry_# Integer
                          Class Longword
                          Size Integer)
                 Returns (Success_Code Integer)

 Entry
 Do
   If KST[Mentor_Segment_#].ASTE_# = Null
   Then Success_Code := Mentor_Seg_Not_Found
        Exit
   Fi
   If Non_Disc_Security(Process_Class,
            KST[Mentor_Segment_#].Class) <> Equal
   Then Success_Code := Not_Allowed
        Exit
   Fi
   Compat_Check := Non_Disc_Security(Class,
                              KST[Mentor_Segment_#].Class)
   If Compat_Check = Less_Than
   Orif Compat_Check = Not_Related
   Then Success_Code := Not_Compatible
        Exit
   Fi
   If Size > Max_Segment_Size
   Then Success_Code := Segment_Too_Large
        Exit
   Fi
   Signal (Memory_Manager,Create_Entry,
           KST [Mentor_Segment_#].ASTE_#,Entry_#,Class,Size)
   Success_Code := Wait
 Od

End Create_Segment
```

Figure 7. Create_Segment Procedure

to the compatibility check, a check must be made to determine if the process access class is equal to the access class of the Alias_Table since adding an entry implies write permission to the Alias_Table. A check is then made on the Size parameter to insure that it is in the range of 256 bytes to 32K bytes. The maximum size of a segment is determined by the size of the design of the secondary storage page table and the hardware constrains the segment to multiples of 256.

If an error is discovered during any of the preceding checks, then an appropriate error code is returned (e.g., Parent_Segment_Not_Found). If there are no errors, the Segment_Manager Signals the Memory_Manager with a request to make an entry in the Alias_Table. The Segment_Manager must Wait for a success code from the Memory_Manager since the Entry_# can only be checked for a duplication by the Memory_Manager. When the Memory_Manager Signals the Segment_Manager that the task has been completed, the Segment_Manager returns the Success_Code to the process. Note that the segment has only been created and if the Supervisor now wishes to reference the segment it must first request the segment be entered into the KST (Make_Known).

Delete_Segment (figure 8) accomplishes the reverse of Create_Segment, that is the removal of a directory entry. The two input parameters for Delete_Segment are Mentor_Segment_# and Entry_#. Again,

```
Delete_Segment Procedure (Mentor_Segment_# Integer
                          Entry_# Integer)
               Returns (Success_Code Integer)

  Entry
  Do
    If KST[Mentor_Segment_#].ASTE_# = Null
    Then Success_Code := Mentor_Seg_Not_Found
         Exit
    Fi
    If Non_Disc_Security(Process_Class,
                KST[Mentor_Segment_#].Class) = Equal
    Then Signal (Memory_Manager,Delete_Entry,
                 KST [Mentor_Segment_#].ASTE_#,Entry_#)
         Success_Code := Wait
    Else Success_Code := Not_Allowed
    Fi
  Od

End Delete_Segment
```

Figure 8. Delete_Segent Procedure

the mentor segment must be known before the Segment_Manager can honor the request. Since the mentor segment must be known, compatibility was checked when the segment was created. The process access class must also be equal to the access class of the mentor segment since deleting an entry implies write permission. When all security checks have been made, the Segment_Manager Signals the Memory_Manager to delete the entry from the Alias_Table. The Segment_Manager Waits for the Memory_Manager to complete the task and it returns the Success_Code from the Memory_Manager to the Supervisor process. The Wait is necessary because an error occurs if the Mentor_Segment is not empty prior to the deletion.

c. Managing the Segmented Address Space

A process must declare an intention to use a segment before it can reference the segment. This declaration introduces the segment into the address space of the process. The way the Supervisor declares its intention to use a segment is to ask that a Segment_# be assigned. This results in an entry in the Known_Segment_Table. Make_Known is the entry point into the Segment_Manager to accomplish an entry in the KST.

A call to Make_Known (figure 9) requires three parameters: 1) Mentor_Segment_#, 2) Entry_#, and 3) Access_Mode_Desired. Segment_# is the value which the Segment_Manager returns to the Supervisor process and is the index to the KST entry and to the segment descriptor

```
Make_Known Procedure  (Mentor_Segment_# Integer
                       Entry_# Integer
                       Access_Mode_Desired Access_Mode)

            Returns (Segment_# Integer
                     Access_Mode_Allowed Access_Mode
                     Success_Code Integer)

Local Index Integer
      ASTE_# Word
      Class Longword
      Size Integer

Entry
  Get_Seg_#: Do
     If KST[Mentor_Segment_#].ASTE# = Null
     Then Success_Code := Mentor_Not_Known
          Exit From Get_Seg_#
     Else Signal (Memory_Manager,Activate,
            KST [Mentor_Segment_#].ASTE_#,Entry_#)
       ASTE_#, Class, Size, Success_Code := Wait
       If Success_Code = Segment_Found
       Then Index := 0
            Search: Do
                      If KST [Index].ASTE_# = ASTE_#
                      Then Segment_# := Index
                           Success_Code := Already_Known
                           Access_Mode_Allowed :=
                                KST[Segment_#].Access_Mode
                           Exit From Get_Seg_#
                      Fi
                      Index += 1
                      If Index > Max_Number_Of_Segments
                      Then Exit From Search
                     Fi
                     Repeat From Search
            Od      !Search!
```

Figure 9. Make_Known Procedure

```
            Index := 0
            Find_Entry: Do
                If KST [Index].ASTE_# = Null
                Then If Non_Disc_Security(Process_Class,
                          Class) = Less_Than
                     Orif Non_Disc_Security(Process_Class,
                          Class) = Not_Related
                     Then Access_Mode_Allowed := Null
                     Else If Non_Disc_Security(Process_Class,
                               Class) = Equal
                          Then Access_Mode_Allowed :=
                                    Access_Mode_Desired
                            Else Access_Mode_Allowed := Read
                            Fi
                      Fi
                      If Access_Mode_Allowed <> Null
                      Then Segment_# := Index
                           KST[Segment_#].ASTE_# := ASTE_#
                           KST[Segment_#].Class := Class
                           KST[Segment_#].Access_Mode :=
                                        Access_Mode_Allowed
                           KST[Segment_#].Size := Size
                           KST[Segment_#].In_Core := No
                           Success_Code := Segment_Found
                           Inner_TC(Add_Seg,Segment_#,
                            Access_Mode_Allowed)
                      Else Segment_# := Null
                           Success_Code := Not_Allowed
                      Fi
                      Exit From Find_Entry
                Fi
                Index += 1
                If Index > Max_Number_Of_Segments
                Then Segment_# := No_Segments_Avail
                     Exit From Get_Seg_#
                Fi
                Repeat From Find_Entry
            Od    !Find_Entry!
Od      !Get_Seg_#!

End  Make_Known
```

Figure 9. Make_Known Procedure (Continued)

in the MMU hardware. Different processes using the same segment will not have the same Segment_# for the segment, since each process has its own KST. Three parameters are returned from Make_Known: 1) the assigned Segment_#, 2) the Access_Mode_Allowed which may be less than Access_Mode_Requested, and 3) a Success_Code. If the Success_Code indicates an error the first two parameters are Null.

Make_Known first Signals the Memory_Manager and Waits for the ASTE_# of the segment. If more than two rings were implemented, ring brackets would also be required from the Memory_Manager [10]. A search of the KST then will reveal if the segment is already known. If it is known, the assigned Segment_# the Access_Mode_Allowed (unchanged), and a Success_Code of Already_Known are returned. Access_Mode_Allowed cannot be changed for segments in the address space. If there is no entry in the KST, an entry is made by filling in the columns of the KST at the first available Segment_#. Non_Discretionary_Security is called to interpret the security labels of the subject and the object. Access to the segment is then granted with the access allowed equal to the less privileged of Access_Mode_Desired or Max_Access_Allowable. If write access is requested but security allows only read, read is the access granted. A call must also be made to the Inner_Traffic_Controller to add the segment descriptor to the hardware descriptor list

(MMU) and the software image of the descriptor list.

If the maximum number of segments is exceeded Make_Known will return No_Segment_Available. The process then has the option of terminating any other segment to make room for the required segment. (Note that the maximum number of segments allowed by the hardware could be exceeded without using all of the linear "virtual core" allocation or conversely.) Terminate is the entry point in the Segment_Manager to remove a segment from the KST.

Terminate (figure 10) is responsible for removing the segment from the address space and reflects this by removing the entry from the KST. The only argument which must be passed is the Segment_# to be terminated. The return argument is a Success_Code. There are four errors which can be found by the Segment_Manager: 1) a segment which is not known, 2) attempting to terminate a segment still loaded in the process virtual core, 3) attempting to terminate a Kernel segment, and 4) passing an invalid Segment_# (too large). The Memory_Manager is Signaled to Deactivate the segment (remove the AST entry) and a Wait occurs until the Deactivate is completed. Note that the Wait is to insure that a race condition between the Memory_Manager and Supervisor process [11] does not occur. The KST entry is deleted by setting the AST_# of the KST entry to null, calling the Inner_Traffic_Controller to delete the segment from the descriptor segment and returning.

```
Terminate Procedure (Segment_# Integer)
          Returns (Success_Code Integer)

  Entry
    Do
      If KST[Segment_#].ASTE_# = Null
      Then Success_Code := Segment_Not_Known
           Exit
      Fi
      If KST[Segment_#].In_Core = Yes
      Then Success_Code := Segment_In_Core
           Exit
      Fi
      If Segment_# <= Number_Kernel_Segments
      Then Success_Code := Kernel_Segment
           Exit
      Fi
      If Segment_# > Max_Segment_#
      Then Success_Code := Invalid_Segment_#
           Exit
      Fi
      Signal(Memory_Manager,Deactivate,KST[Segment_#].ASTE_#)
      Success_Code := Wait
      KST[Segment_#].ASTE_# := Null
      Inner_TC(Delete_Seg,Segment_#)
    Od

End Terminate
```

Figure 10. Terminate Procedure

d. Moving Segments into Memory

Swap_In (figure 11) and Swap_Out (figure 12) are the two procedures in the Segment_Manager which move segments between main memory and secondary storage. (Secondary storage is used as a generic term in this thesis to indicate all memory of a computer system other than main or core memory. It includes "tertiary" or lower order memory.) To move a segment from secondary storage to main memory, a process must call Swap_In with the Segment_# and Base_Address as arguments. Base_Address is the location in the linear virtual core of the process where the segment is to begin. This is a virtual core address and does not correspond to a real address in memory; in fact, memory cannot be addressed at all except by addressing a segment. The Segment_Manager indexes to the segment in the KST to retrieve the necessary attributes for moving the segment. If the segment is not found, Segment_Not_Found is returned. After obtaining the attributes of the segment, the Segment_Manager Signals the Memory_Manager to do the transfer. A Wait is then executed until the Memory_Manager can send the Absolute_Address in real memory to the Segment_Manager. This information is passed to the Inner_Traffic_Controller to update the absolute address in the hardware and software descriptor lists. This procedure only works because of the design choice not to unload a process from a virtual processor. If processes are unloaded the Memory_Manager would have to

```
Swap_In Procedure (Segment_# Integer
                   Base_Address Word)
        Returns (Success_Code Integer)

 Entry
   If KST[Segment_#].ASTE_# = Null
   Then Success_Code := Seg_Not_Found
        Exit
   Fi
   Signal (Memory_Manager,In,Segment_#,
           KST [Segment_#].ASTE_#,Base_Address,
           KST[Segment_#]>Access_Mode)
   Absolute_Address,Success_Code := Wait
   If Success_Code = Swapped_In
   Then Inner_TC (Load,Segment_#,Absolute_Address,
                  KST[Segment_#].Size)
        KST[Segment_#].In_Core := Yes
   Fi
End Swap_In
```

Figure 11. Swap_In Procedure

```
Swap_Out Procedure (Segment_# Integer)
         Returns (Success_Code Integer)

  Entry
    If KST[Segment_#].ASTE_# = Null
    Then Success_Code := Seg_Not_Found
         Exit
    Fi
    Written := Inner_TC (Unload,Segment_#)
    Signal (Memory_Manager,Out,KST[Segment_#].ASTE_#,Written)
    KST[Segment_#].In_Core := No
    Success_Code := Swapped_Out

End Swap_Out
```

Figure 12. Swap_Out Procedure

call the Inner_Traffic_Controller. The parameter returned to the process indicates if the segment swap-in was successful.

The move in the other direction--main memory to secondary storage--is performed by Swap_Out. The only input argument is the Segment_# and Success_Code is the only return argument. After validation of the Segment_#, the Segment_Manager calls the Inner_Traffic_Controller to obtain the status of the hardware changed bit. This is in turn passed by Signal to the Memory_Manager to make the change. Success_Code is set to Swap_Out and the Segment_Manager returns. If more than one processor is used in the system, race conditions should be investigated in this procedure of allowing the Segment_Manager rather than the Memory_Manager to call the Inner_Traffic_Controller.

To this point the usual order for invoking the Segment_Manager functions has not been specified. There is a usual sequence of events. In order: Create_Segment to make an Alias_Table entry, Make_Known to introduce the segment into the address space, and Swap_In to move the segment into the process's virtual core are the steps necessary before a process can make a reference to a segment. Conversely, Swap_Out, Terminate, and Delete_Segment is the order to move a segment from main memory to secondary storage, remove the entry from the KST and descriptor from the MMU, and remove the segment from

the address space. If the functions are called in any other order, the usual result is an error condition and no action is taken. No "harm" results from calls made out of sequence.

5. Traffic Controller Module

The Traffic_Controller is responsible for multiplexing processes onto virtual processors. A virtual processor is an abstraction which describes a logical processor. There are multiple virtual processors which exist on a single physical processor. The Traffic_Controller is also the Kernel module which supports the interprocess communication primitives, Block and Wake_Up. In the Archival Storage System, Block and Wake_Up are the last two of the six user entries into the Kernel. There are four other procedures in the Traffic_Controller which implement the scheduling algorithm and provide message queue services for Block and Wake_Up.

a. Active Process Table

The database of the Traffic_Controller is the Active Process Table (APT) (figure 13). This is a fixed-size table in the Kernel because of the decision not to create or destroy processes. When the Archival Storage System goes through system generation, each process will be created and an entry made in the APT. The process will then be active for the life of the system. Each active process will have a unique identifier (Process_ID) which

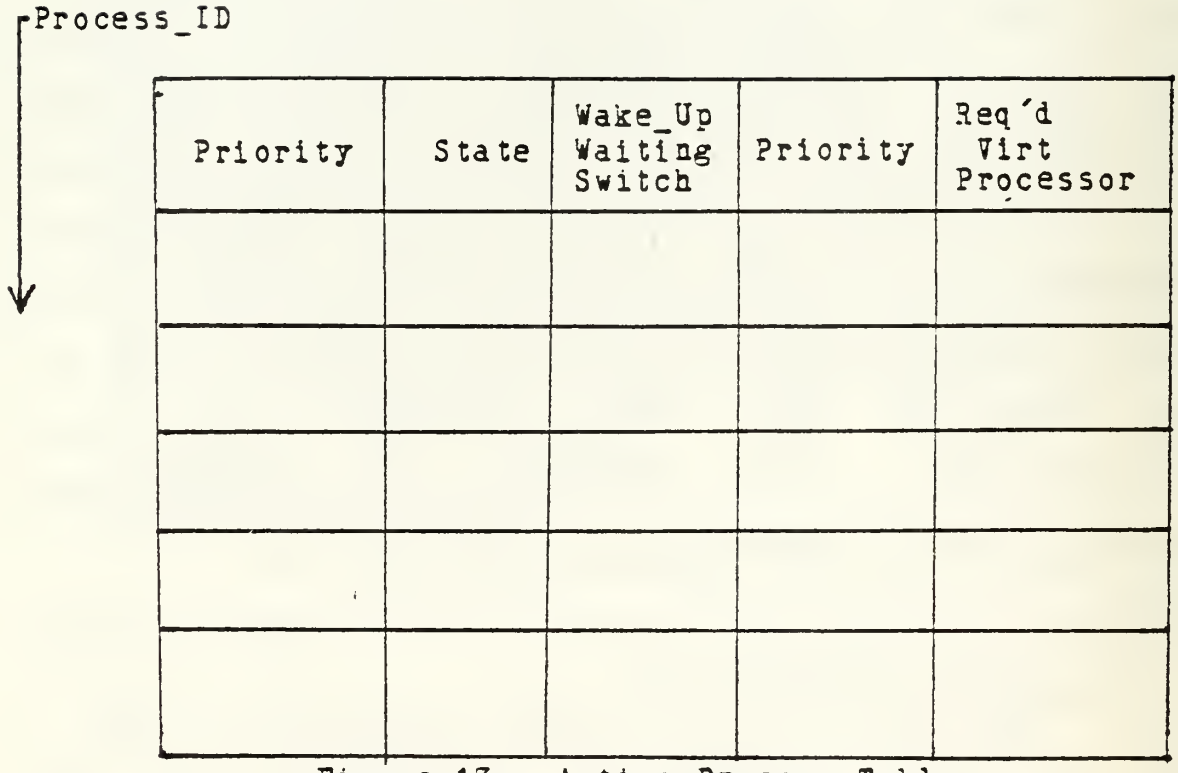


Figure 13. Active Process Table

is also the index to the APT. Note that if processes were created and destroyed, then allowing Process_IDs to leave the Kernel could create a communication path. In that case the Process_ID should be "virtualized". The State field of the APT indicates whether a process is blocked, ready, or running.

An explanation of the interprocess communication primitives is necessary here. Block and Wake_Up [19] are the interprocess communication primitives used by cooperating processes in the Supervisor domain. Invocation of the primitives is actually a call to the Traffic_Controller and causes the Traffic_Controller to execute the scheduling algorithm. A process calls Wake_Up when it has a message or task for another process. Wake_Up will set the state of a blocked process to ready. If the process is ready or running it will have no effect on the status of the process. When a process cannot continue execution until a reply to a Wake_Up is received, the process must block itself. Block will set the process status to blocked.

Within the Kernel Signal and Wait are the primitives used for communication. They function in the same manner as Block and Wake_up, but are calls to the Inner_Traffic_Controller instead of the Traffic_Controller. Signal and Wait are bounded in time which indicates that they are guaranteed to return. Block and Wake_Up are not bounded since no claims can be made

about correctness of calls from outside the Kernel. It is possible for a user process to call Block erroneously and never be heard from again.

The Wake-Up Waiting Switch is Saltzer's [19] mechanism for synchronization of interprocess communication primitives. Without the switch a race condition can occur. For example, the following sequence of events could happen because processes can run simultaneously:

1) Process A looks in its work queue and finds it empty.
2) Process B puts a task in A's work queue.
3) B wakes up A.
4) A blocks itself.

At step 3, A was running, so the wake-up sent by B was ignored. When A called block, a task is in the work queue, but A missed the wake-up signal, so the task remains uncompleted. In particular, if A was expecting some event necessary for A to continue, A may never wake-up.

The Wake-Up Waiting Switch prevents the occurrence of such a situation by requiring the following sequence of actions:

Process B:

1) Process B puts task in Process A's work queue.
2) Wake-up A and turn wake-up waiting switch on.

Process A:

1) Reset the wake-up waiting switch to off.
2) Look in the work queue and find it empty.
3) Call Block, which returns if wake-up waiting switch is on.

Now, the above sequences can occur in any time relationship and the wake-up signal will have the desired effect.

The Traffic_Controller uses the priority field for determining what process to schedule to run on the virtual processor. The Required_Virtual_Processor field is used to bind a loaded process to a specific virtual processor. Only two processes run on a virtual processor—the loaded process and the "Idle" process. This is a direct result of the simplifying design choice (to have all processes loaded) made for the Archival Storage System. In general, processes must be loaded and unloaded. The Idle process is put into the running state whenever the loaded process blocks itself.

b. Interprocess Communication Primitives

Because the Archival Storage System does not allow creation or destruction of processes except at system generation, the only external entry points into the Traffic_Controller are Block and Wake_Up. As previously explained, Block and Wake_Up are the primitives used by Supervisor processes for interprocess communication.

Block (figure 14) is called when a process

```
Block Procedure
      Returns (Process_ID Integer
               Message Message_Type)

Entry
  If APT [Process_ID].Wakeup_Waiting_Switch = On
  Then APT [Process_ID].Wakeup_Waiting_Switch := Off
  Else APT [Process_ID].State := Blocked
       Sched_Ready_Process
  Fi
  Process_ID,Message := Get_First_Message(Message_Queue)

End Block
```

Figure 14. Block Procedure

cannot continue until the occurrence of some other event. After going through the Gate_Keeper, the call enters the Traffic_Controller. The Wake_Up_Waiting_Switch is immediately checked. If the switch is on, the switch is reset to off, and the first message in the Message_Queue for the process is retrieved. The Traffic_Controller then returns through the Gate_Keeper.

If the Wake_Up_Waiting_Switch was off then the state of the process is set to Blocked. Sched_Ready_Process is called to schedule the highest priority ready process on the virtual processor. In the Archival Storage System this is a trivial task, because the only other process which is loaded on the virtual processor is the idle process. The idle process can never block itself, so it must always be either running or ready. In fact the idle process will only consist of a halt instruction.

The Traffic_Controller could have been collapsed into the Inner_Traffic_Controller for this design, but preservation of generality was a design goal. Later extensions will be easier to implement since the basic structure of the Traffic_Controller is present.

The counterpart of Block is Wake_Up. Wake_Up (figure 15) is used by processes in the Supervisor domain to pass messages to other processes in the Supervisor domain. Upon entry into Wake_Up, the message is placed in the Message Queue of the awakened process. The

```
Wake_Up Procedure (Wakeup_Process_ID Integer
                   Message Message_Type)
        Returns (Success_Code Integer)

Entry
  Do
    Success_Code := Insert_Message(Wakeup_Process_ID Message)
    If Success_Code = Queue_Overflow
    Orif Success_Code = Not_Allowed
    Then Exit
    Else APT [Wakeup_Process_ID].Wakeup_Waiting_Switch := On
         If APT [Wakeup_Process_ID].State = Blocked
         Then APT [Wakeup_Process_ID].State := Ready
              Enter_Ready_Queue (Wakeup_Process_ID)
         Fi
         APT [Process_ID].State = Ready
         Enter_Ready_Queue(Process_ID)
         Sched_Ready_Process

     Fi
   Od
End Wake_Up
```

Figure 15. Wake-up Procedure

Wake_Up_Waiting_Switch of the process to be awakened is then set to On. Then if the process state is blocked it is put into the Ready_Queue and the State is set to ready. Regardless of the state of the awakened process, the waking process then puts itself into a Ready State and Enters the Ready_Queue itself. This is necessary because the process to be awakened may have a higher priority than the waking process. Every time either Block or Wake_Up is called the scheduling algorithm is executed (Sched_Ready_Process).

c. Process Scheduling Algorithm

Enter_Ready_Queue (figure 16) and Sched_Ready_Process (figure 17) are two internal functions of the Traffic_Controller. Enter_Ready_Queue is used for placing a ready process into a first-in, first-out queue which is organized by priority (figure 18). The Ready_Queue is designed as a two-dimensional array indexed by Priority and a top and bottom pointer. The algorithms for all queue operations are taken from Knuth [21]. When a Process_ID is to be added to the queue the bottom pointer for the appropriate priority queue is incremented by one. If the bottom pointer is at the bottom of the linear array which implements the queue then it is set to the first location of the array, thus wrapping around. The physical length of each queue column is equal to the total number of processes which can be entered into that queue at any point in time so that the queue cannot overflow. The

```
Enter_Ready_Queue Procedure (Process_ID Integer)

Entry
  If Ready_Queue_Bottom [APT [Process_ID].Priority] =
     Max_Queue_Length
  Then Ready_Queue_Bottom [APT [Process_ID].Priority] := 0
  Else Ready_Queue_Bottom [APT [Process_ID].Priority] += 1
  Fi
  Ready_Queue[APT [Process_ID].Priority, Ready_Queue_Bottom]
                  := Process_ID

End Enter_Ready_Queue
```

Figure 16. Enter_Ready_Queue Procedure

```
Sched_Ready_Process Procedure

Entry
  Priority := Max_Priority
  Scan: Do
    If Ready_Queue_Top [Priority] =
       Ready_Queue_Bottom [Priority]
    Then Priority -= 1
         If Priority < Min_Priority
         Then Exit From Scan
         Else Repeat From Scan
         Fi
    Else If Ready_Queue_Top[Priority] = Max_Queue_Length
         Then Ready_Queue_Top[Priority] := Ø
         Else Ready_Queue_Top[Priority] += 1
         Fi
      Run: If APT[Ready_Process_ID].Reqd_Virt_Processor
                = Processor_ID
           Then APT [Ready_Process_ID].State := Running
                Inner_TC (Swap_MMU,Ready_Process_ID)
           Else Get_Next_Process(Ready_Queue)
                Repeat From Run
           Fi
    Fi
  Od

End Sched_Ready_Process
```

Figure 17. Sched_Ready_Process Procedure

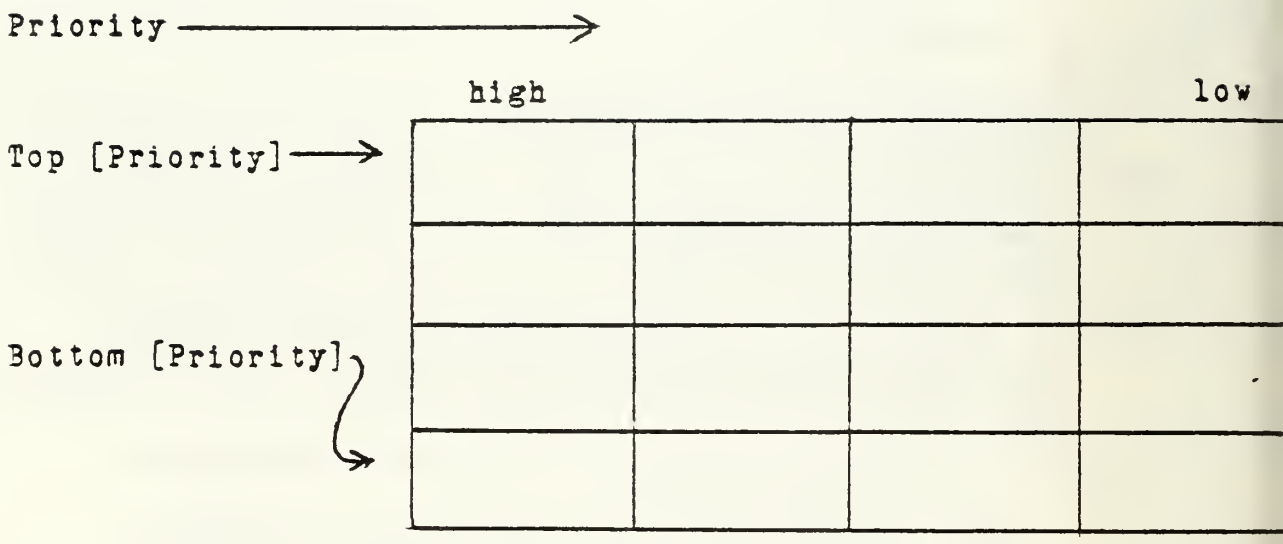


Figure 18. Ready Queue

Process_ID is placed into the array at the location pointed to by the bottom pointer. The queue is always entered at the logical bottom and removal takes place from the logical top.

The procedure which removes the processes from the top of the queue is Sched_Ready_Process. The function of Sched_Ready_Process is to "pass" (as a baton in a relay race) the current virtual processor to the highest priority, ready process which can run on this specific virtual processor. Starting with the Max_Priority queue, each queue is scanned until the first ready process that can run on the virtual processor currently executing in the Traffic_Controller is encountered. Each queue is tested in turn to determine if it is empty. If the queue is empty, then the next lower priority queue is scanned. The existence of an Idle process for each virtual processor guarantees that a ready process is always found, so the Traffic_Controller cannot exit without scheduling a process. When a ready process is found, then the process State is set to running (scheduled) and the Inner_Traffic_Controller is called to Swap_MMU. This generally will load the process descriptor segments into the Virtual_Processor_MMU, but in the design of the Archival Storage System the MMU of the Idle process is identical to the MMU of the loaded process.

d. Message Queue Operators

The Message_Queue (figure 19) is a

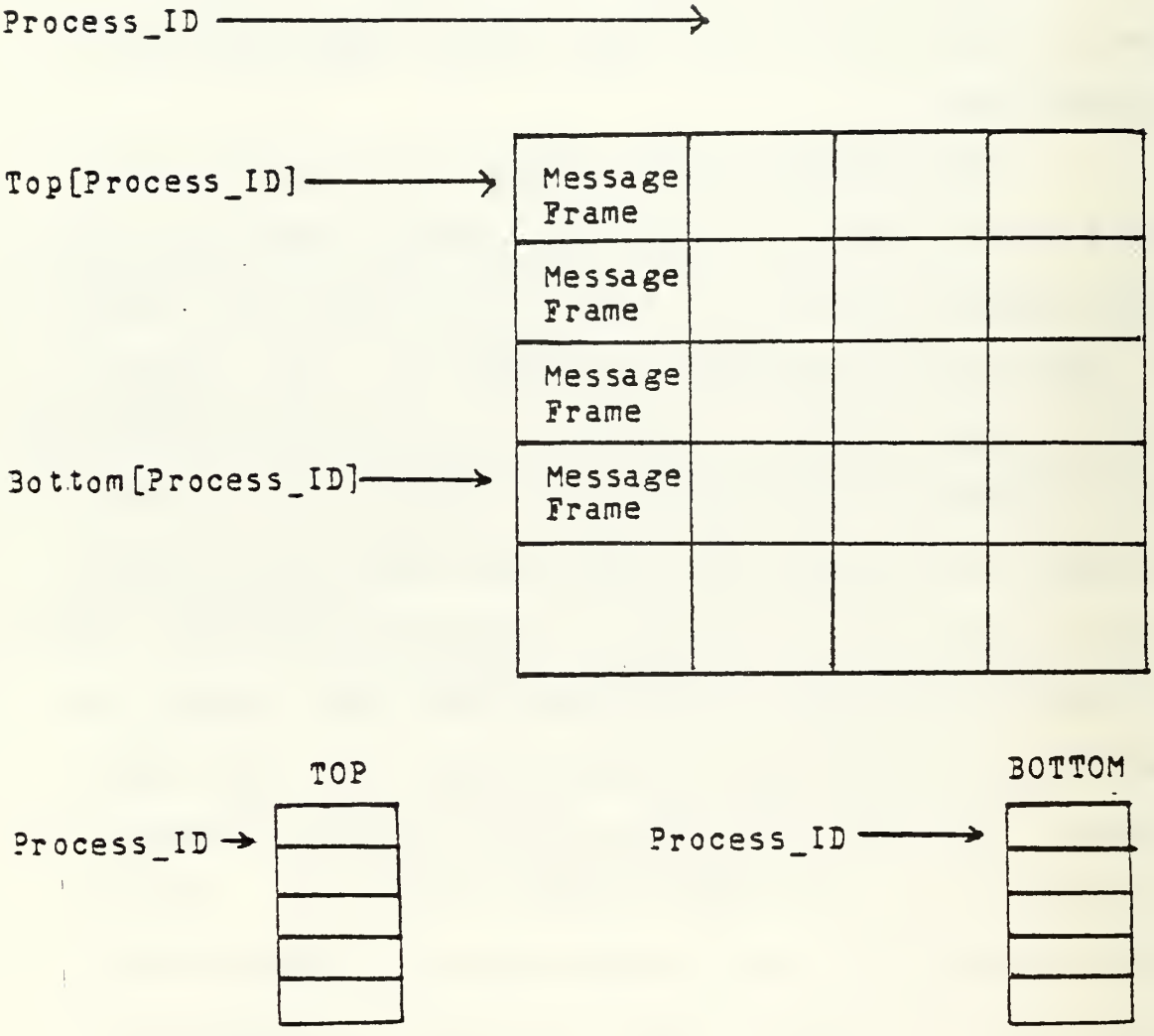


Figure 19. Message Queue And Pointers

two-dimensional array of message "frames". It is indexed in one dimension by the Process_ID and in the other dimension by a top and bottom pointer. Insert_Message (figure 20) is the primitive used by Wake_Up to put a message into another process' message queue. The design only allows communication between processes of equal security class since a Success_Code is returned to the waking process. Get_First_Message (figure 21) is the primitive used by Block to retrieve messages from the message queue. If the queue is empty, the message "Queue_Empty" is returned.

6. Non-Discretionary Security Module

The key to implementing a particular non-discretionary security policy is in one module. By representing the policy as a partially ordered lattice, an interpretation algorithm can be written to make a comparison between two labels and return a relationship. The relationship can be equal, less than, greater than, or not related.

The Non_Discretionary_Security Module shown in figure 22 will determine the relationship of three categories of classification (Secret, Confidential, Unclassified). As shown there are no checks for compartments (e.g., crypto, nuclear, etc.). If a complete DOD security policy interpretation is desired, the module can be expanded. Since some DOD specifications require provisions for eight categories and sixteen compartments,

```
Insert_Message Procedure (Message_Queue_ID Integer
                          Message Message_Type)
                Returns (Success_Code Integer)

Entry
  If Non_Disc_Security (APT[Process_ID].Class,
            APT[Message_Queue_ID].Class) = Equal
  Then
  If Message_Queue_Bottom[Message_Queue_ID]
                  = Max_Queue_Length
  Then If Message_Queue_Top[Message_Queue_ID] = Ø
       Then Success_Code := Queue_Overflow
       Else Message_Queue_Bottom[Message_Queue_ID] := Ø
            Message_Queue[Message_Queue_Id,
            Message_Queue_Bottom[Message_Queue_ID]]
                              := Message,Process_ID
            Success_Code := Inserted
       Fi
  Else If Message_Queue_Bottom[Message_Queue_ID] + 1
                 = Message_Queue_Top[Message_Queue_ID]
       Then Success_Code := Queue_Overflow
       Else Message_Queue_Bottom[Message_Queue_ID] += 1
            Message_Queue[Message_Queue_ID,
            Message_Queue_Bottom[Message_Queue_ID]]
                          := Message,Process_ID
            Success_Code := Inserted
       Fi
  Fi
  Else Success_Code := Not_Allowed
  Fi
End Insert_Message
```

Figure 20. Insert_Message Procedure

```
Get_First_Message Procedure (Message_Queue_ID Integer)
                  Returns (First_Message Message_Type)

Entry
  If Message_Queue_Top[Message_Queue_ID] =
       Message_Queue_Bottom[Message_Queue_ID]
  Then First_Message := Queue_Empty
  Else If Message_Queue_Top[Message_Queue_ID] =
            Max_Queue_Length
       Then Message_Queue_Top[Message_Queue_ID] := 0
       Else Message_Queue_Top[Message_Queue_ID] += 1
       Fi
       First_Message := Message_Queue[Message_Queue_ID,
                  Message_Queue_Top[Message_Queue_ID]]
  Fi

End First_Message
```

Figure 21. Get_First_Message Procedure

```
Non_Disc_Security Procedure (Class_1 Longword
                             Class_2 Longword)
                    Returns (Relationship Integer)

Entry
  If Class_1
    Case Unclassified Then
      If Class_2
        Case Unclassified Then Relationship := Equal
        Case Confidential,Secret Then Relationship
                              := Less_Than
        Else Relationship := Not_Related
      Fi
    Case Confidential Then
      If Class_2
        Case Unclassified Then Relationship := Greater_Than
        Case Confidential Then Relationship := Equal
        Case Secret Then Relationship := Less_Than
        Else Relationship := Not_Related
      Fi
    Case Secret Then
      If Class_2
        Case Unclassified,Confidential Then
                    Relationship := Greater_Than
        Case Secret Then Relationship := Equal
        Else Relationship := Not_Related
      Fi
    Else Relationship := Not_Related
  Fi

End Non_Disc_Security
```

Figure 22. Non_Disc_Security Procedure

a longword was chosen as the data type for representing the labels. The 32 bits of a longword are more than sufficient to represent all possible combinations of categories and compartments.

Similarly, Privacy Act requirements are easily implemented in Non_Discretionary_Security since they can be represented by a lattice structure. Most other practical non-discretionary security policies can be implemented as well.

7. Inner Traffic Controller Module

The Inner_Traffic_Controller provides the multiplexing of virtual processors to the real processor of the system. Each loaded process will be allocated to a virtual processor, implying that there is a many to one correspondence. In order to manage these virtual processors, the Inner_Traffic_Controller has direct access to the machine hardware. The Memory Management Unit and processor state are loaded and unloaded by the Inner_Traffic_Controller, thus accomplishing the multiplexing to the physical processor.

In addition to managing the virtual processors, the Inner_Traffic_Controller furnishes inter-process services. Signal and Wait are used by processes in the Kernel ring to communicate with other Kernel ring processes and are primitives of the Inner_Traffic_Controller.

The main database used to handle the

Inner_Traffic_Controller functions is the Virtual_Processor_Table. Additionally, a software image of each MMU is maintained for every loaded process.

a. Virtual Processor Table

The Virtual_Processor_Table (figure 23) is indexed by the Virtual_Processor-ID. Each virtual processor can be in one of three states: 1) Running, 2) Ready, or 3) Waiting. These three states are analogous to the state of a process and are used for processor scheduling in the same manner as the Traffic_Controller used the state of a process for scheduling processes. After the State field is the Signal_Pending_Switch which functions precisely as the Wake_Up_Waiting_Switch for preventing a race condition from occurring with the interprocess communication primitives. Priority is the next field which is also analogous to the APT priority.

Loc_Processor_State is a pointer to the area in memory where the MMU software image is maintained as well as the "save block" for the machine state of the virtual processor when it is ready or waiting. Figure 24 is an example of the format of the MMU image.

b. Kernel Interprocess Communication Primitives

Signal and Wait function in the same manner as Block and Wake-Up. The chief distinction between the pairs is the degree of trust placed on the correctness of use. Since Signal and Wait are Kernel primitives which are used only by process operating in the Kernel domain, the calls

Virt_
Processor_
ID

State	Signal Pending Switch	Priority	Loc_ Processor Image

Figure 23. Virtual Processor Table

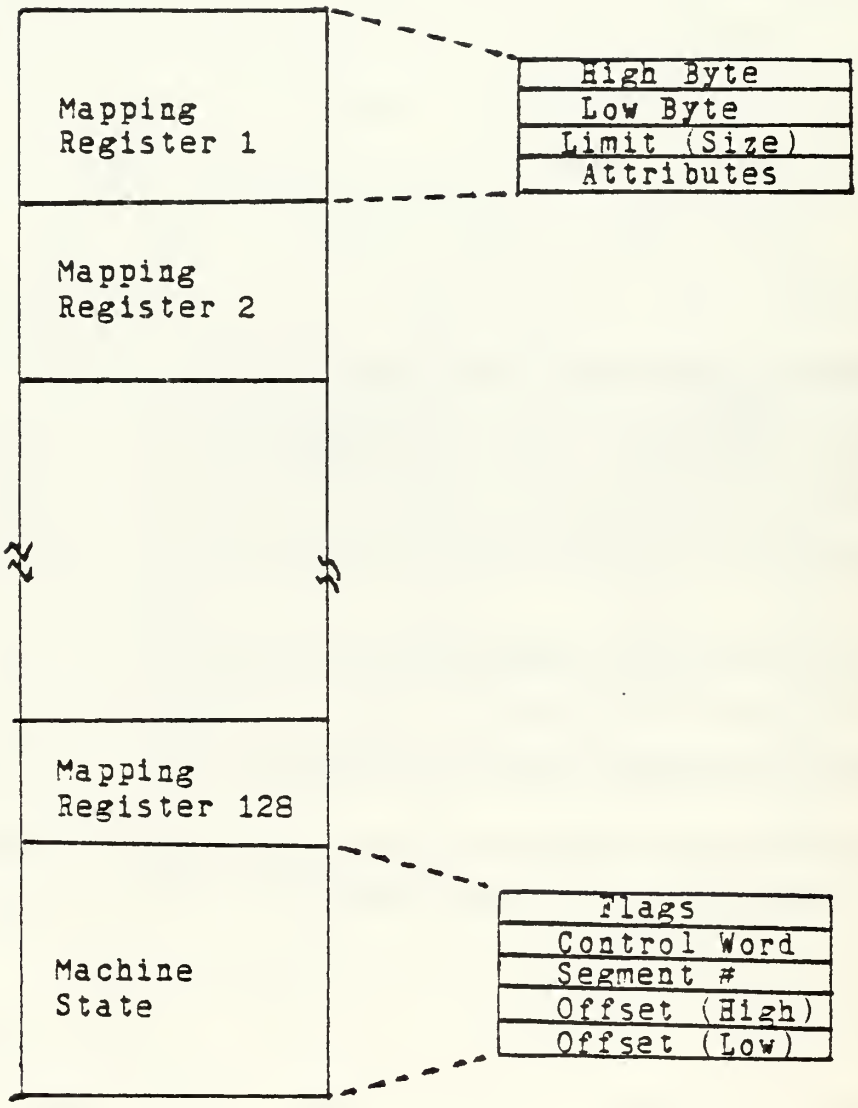


Figure 24. MMU Image

can be guaranteed to return. The same trust cannot be placed on the calls to Block and Wake_Up by processes in the Supervisor ring. The loop free structure implies that the Kernel neither knows nor cares what happens in the outer domain (or domains, if present). Yet, the Kernel must not allow the security state of the machine to change except in accordance with the rules of the mathematical model. Block is restricted to communication among processes at the same level. The Kernel must call upon processes operating at different security levels to accomplish its task and thus needs a different primitive since systemwide information is being passed.

With one exception, Signal (figure 25) and Wait (figure 26) function in the same manner as Wake_Up and Block do in the Traffic_Controller. Since the data structures in the Inner_Traffic_Controller function with virtual processors, the Signaled_Process_ID or Process_ID (input parameters) must be translated into a Signaled_Processor_ID or Processor_ID. A one-dimensional table is maintained for this purpose. Because the Inner_Traffic_Controller must complete its task before it returns to the calling procedure and is synchronous to the progress of the process, the table translation of process to virtual processor works. The Idle processes will never try to Signal or Wait and will never cause the scheduling algorithm to be executed.

It is possible for the Idle process to be

```
Signal Procedure (Signaled_Processor_ID Integer
                  Signal_Message Message_Type)
        Returns (Success_Code Integer)

Entry
  Do
    Signaled_Processor_ID := Map(Process_ID)
    Success_Code := Insert_Message(Signaled_Processor_ID Messag
    If Success_Code = Queue_Overflow
    Orif Success_Code = Not_Allowed
    Then Exit
    Else VPT [Signaled_Processor_ID].Signal_Pending_Switch := C
          If VPT [Signaled_Processor_ID].State = Waiting
          Then VPT [Signaled_Processor_ID].State := Ready
               Enter_Ready_Queue (Signaled_Processor_ID)
          Fi
          VPT [Processor_ID].State = Ready
          Enter_Ready_Queue(Processor_ID)
          Sched_Ready_Processor

     Fi
   Od
End Signal
```

Figure 25. Signal Procedure

```
Wait Procedure
      Returns (Process_ID Integer
               Signal_Message Message_Type)

Entry
  Processor_ID := Map(Process_ID)
  If VPT [Processor_ID].Signal_Pending_Switch = On
  Then VPT [Processor_ID].Signal_Pending_Switch := Off
  Else VPT [Processor_ID].State := Waiting
       Sched_Ready_Processor
  Fi
  Signal_Message := Get_First_Sig_Mess(Sig_Queue[Processor_ID])

End Wait
```

Figure 26. Wait Procedure

scheduled on each virtual processor in the storage system. When that occurs the real processor will come to a standstill, executing a Halt instruction. At first glance this would seem to be an error condition, but in reality it is not. Since the Archival System is driven by external events this may at times be a normal state. When a request is made from a Host, the interrupt handler (an I/O_Manager entry) will Signal (via the Inner_Traffic_Controller) the appropriate process and cause the scheduling algorithm to be executed.

c. Service Functions

All of the functions of the Inner_Traffic_Controller are called from the Kernel ring. Add_Seg, Delete_Seg, Load, and Unload are service calls to support the Segment_Manager. These are hardware dependent functions and the details of their design will be influenced by the specific characteristics of the MMU and CPU hardware. Add_Seg makes an entry into an MMU hardware descriptor and also the MMU software image. This call is made from Make_Known and will only set up the descriptor. Since the segment has not been Swapped_In at this point, the address fields of the descriptor will be null and the attribute field of the descriptor will be set to inhibit the CPU from making access.

Delete_Seg is called from terminate and is required to remove an entry from the MMU and the software image. Load will place the absolute location of the

segment base address into the MMU and change the attributes to allow the CPU access. Unload removes the segment base address, inhibits CPU access again and also retrieves the changed bit from the attribute field. This changed bit is set when a segment is written and is used by the Memory_Manager to decide if the segment can be overwritten or if it must be written back to secondary storage. A variant of Load and Unload is needed by the Memory_Manager when doing a local to global move.

Swap_MMU is called from the Traffic_Controller and is a result of the scheduling algorithm being executed. In the general case a process swap would occur on the virtual processor as a result of this call. In the Archival Storage System, there are only two processes which are allowed to run on a virtual processor: 1) the loaded process or 2) the Idle process. An MMU swap will still occur conceptually when the idle process is loaded because it has an MMU image just as any other process. Actually the idle process's MMU image is exactly the same as the loaded process, so a physical swap does not take place.

Other service calls will be made to the Inner_Traffic_Controller from the Memory_Manager and I/O_Manager but are not detailed here. Software faults, as discussed in O'Connell and Richardson [5], are not needed in this design.

8. Memory Manager Module

The Memory_Manager is a non-distributed Kernel process and is responsible for managing the real memory resources of the system. The real memory of the system is both main memory (random access) and secondary storage (non-random access). The Memory_Manager could be part of the distributed Kernel in the Archival Storage System since it is designed for a single microprocessor; however, the process abstraction is used to maintain the "family member" character of the design.

a. Memory Management Scheme

The two main tasks of the Memory_Manager are to bring segments into memory (In) or remove segments from memory (Out). Partitioned allocation is the scheme employed to manage the memory resource. Each loaded process is given a partition of linear contiguous real core and is required to manage (via calls to Swap_In and Swap_Out) the partition (its linear virtual core) in any way it chooses. The Memory_Manager checks each 'In' request against the process's allocation to insure that the allocation is not exceeded and to insure that previously allocated memory is not overlayed.

When a shared segment is not writeable (i.e., write permission has not been given to any process), the design allows multiple copies (one per process) of the segment to exist. This frees the Memory_Manager from the task of moving the segment to "processor global" memory, requesting that all MMU images be updated, and reserves

global memory for segments which are shared and writeable. Furthermore, the space that can be saved by having one copy would not be usable by the processes which are sharing the segment, since each process's Supervisor would still have the segment in its virtual core.

If a segment is to be shared and is writeable, then the Memory_Manager must move it to global memory [5]. This insures that all users are sharing the same information. Again, the actual location of the segment is invisible to the sharing processes. More memory is allocated to the segment than it actually uses: viz., one copy per sharing process. However, the alternative to using memory is a complex algorithm for dynamically reconfiguring the mapping of each partition whenever a shared segment is relocated in memory. The tradeoff of memory size for complexity is indicated in this application. Segments are placed in memory within the appropriate partition at the location specified by the Supervisor call to Swap_In. A simple bit map known as the Memory_Allocation_Map (figure 27) is used to indicate which parts of memory are available for use. Each bit of the map corresponds to a 256-byte page of memory. The term page is not used here in the classical sense, but is used to indicate a block of physical memory. Segments cannot be divided into pages scattered through core, but must be allocated to contiguous memory locations.

The primary database of the Memory_Manager is

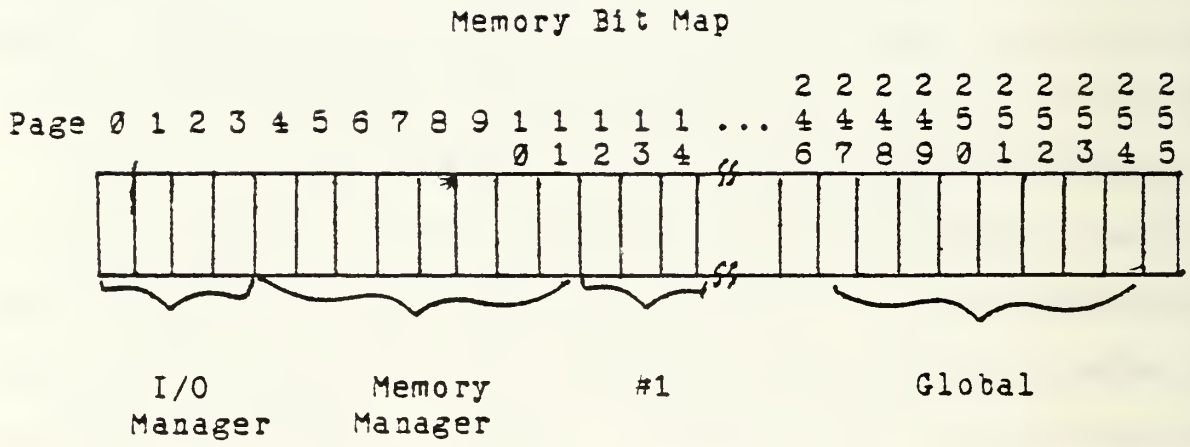


Figure 27. Memory Allocation Map

the Active_Segment_Table. It provides the Memory_Manager with the information necessary for managing all segments in the system which are active.

b. Active Segment Table

There are two sections of the Active_Segment_Table (AST). That portion of the AST which contains systemwide information is known as the Global_Active_Segment_Table (G_AST). Every active segment in the system will have an entry in the G_AST. The Memory_Manager also maintains a portion of the AST per physical processor as the Local_Active_Segment_Table (L_AST). Only those segments active within the physical processor will be in the L_AST.

When a segment is "Made_Known" it becomes active and will have an entry in the G_AST (figure 28) and in the appropriate L_AST (figure 29). The concatenation of the segment's Unique_ID and the index to the segment's entry in the G_AST form the ASTE_# which is the "handle" passed by the Memory_Manager for identifying a specific active segment. When the Memory_Manager uses the "handle" to enter the G_AST, it uses the Entry_# of the ASTE_# portion as the index. In the general case (e.g., demand activation/deactivation), the Unique_ID of the "handle" is then compared with the Unique_ID found in the G_AST entry. If the identification check results in a mismatch, the G_AST must be searched using the Unique_ID as a key to find the correct entry. This procedure is necessary

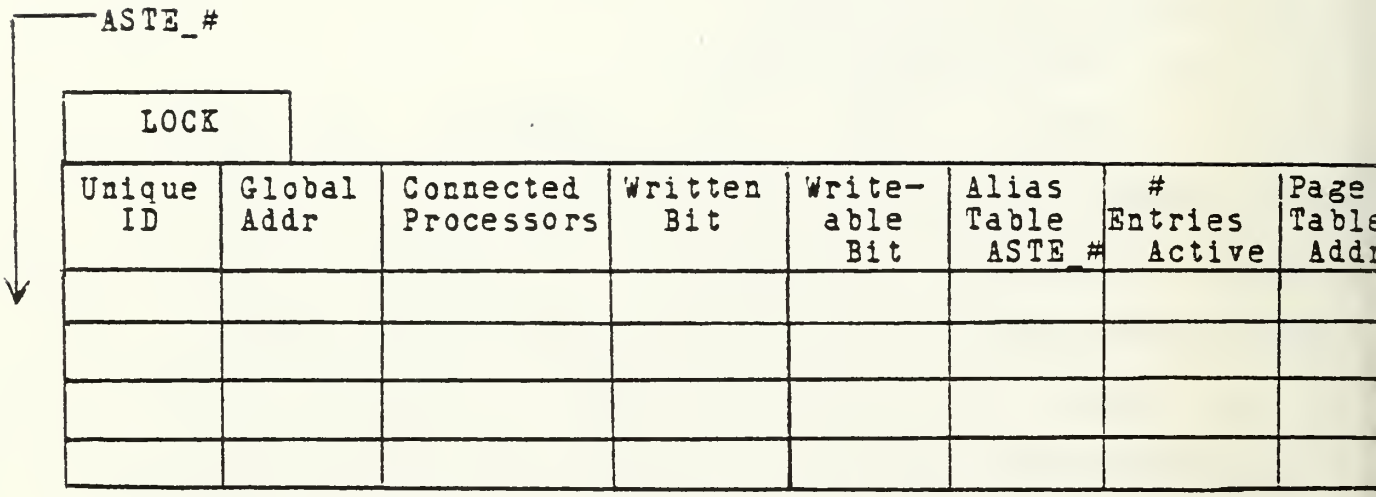


Figure 28. Global Active Segment Table

Unique ID	Access	Absolute Address	Size	Segment #

Figure 29. Local Active Segment Table

because it is possible that a segment's entry could be moved in the G_AST before all processes could be notified of the new ASTE_#. If this occurred and a check was not made, an unauthorized access could then take place. If the match is successful when first checked, the proper entry has been found. In this design all known segments for all processes are active so this problem cannot occur.

Since the G_AST is a systemwide resource a lock must be used on the G_AST to prevent a race condition from occurring [11]. The mechanism used in the design is a locked/unlocked flag. Synchronization on the lock is inherent in the functioning of the Memory_Manager's Signal_Message_Queue. Note that this mechanism will not work if the design is extended to include more than one processor in the system sharing the single G_AST.

The Global_Address field is used only if the segment is located in global memory. If it is null the address can be found in the L_AST. The Connected_Processes field is a bit map signifying which processes currently have the segment active.

The Written flag is used to retain a written bit when a process Swaps_Out a segment which is shared and writeable. For example: Processes A and B are sharing a segment and Process A has write permission. A has written in the segment and now wants to deactivate the segment. Process B is still using the segment. When A requests the Deactivate, the Written bit is passed to the

Memory_Manager. But since B continues to use the segment, the Memory_Manager will only reset Process A's flag in the Connected_Process field. The Written bit is then logically ORed with the G_AST_Written_Flag. When B then Deactivates the segment, the Written bit it passes indicates that a write has not taken place. An error would have occurred if the Written bit from Process A had not been saved since the Memory_Manager does not write an unmodified segment back to secondary storage.

The Writeable flag is set whenever any process has write access to the segment. This is the key flag for deciding (at the time activation is requested) if the segment must be placed in global memory. It cannot conveniently be used to provide an alternative to the scenario presented above for Written. Consider that Processes A, B, and C all have writeable shared access. If A Deactivates after writing, the Memory_Manager could write back to secondary storage at that time, (assuming the proper synchronization was used to prevent B or C from writing while the transfer took place). Then when B or C Deactivated after writing, another write to secondary storage would take place. Thus at least one unnecessary action took place.

The Alias_Table_ASTE_# will be null unless the segment is a mentor segment. Whenever a mentor segment is made active its Alias_Table segment is made active at the same time and will be assigned an ASTE_#. (The Alias_Table

is a Memory_Manager object. The Alias-Table is actually implemented as a collection of segments.)

In the general case with demand activation/deactivation, the #_Entries_Active is a field which is used for Alias_Table entries only. An Alias_Table segment must remain active as long as any of its entries are active, although it need not remain in main memory. The #_Entries_Active is a counter which is incremented any time an Alias_Table Entry is activated and decremented when an Alias_Table Entry is deactivated. Thus the Alias_Table frame can be deactivated only when the Connected_Processor map of the mentor segment and the #_Entries_Active both become zero or null. (Note that the Connected Processor Map of the Alias_Table segment will always show only the physical processors whose Memory_Manager has the Alias_Table in its address space.) In this design all known segments are active so these explicit checks upon deactivation are not required.

The remaining field of the G_AST is the Page_Table_Address. The Page_Table_Address is the location in secondary storage of the page table. The page table in turn provides the location of the segment.

The L_AST portion of the AST is maintained per physical processor and should not be confused with a distributed data structure since the L_AST is a Memory_Manager data structure and not part of the distributed Kernel. It is searched by Virtual_Processor_ID

and segment Unique_ID. The remaining four fields are Access, Absolute_Address, Size, and Segment_#. The Access is the read or read/write access of the segment available for use in moving between local and global memory. The Absolute_Address is the location of the segment in main memory. If Absolute_Address is null, the segment is on secondary storage and has not been moved to main memory.

c. Aliasing Scheme

The Memory_Manager also provides the aliasing service for the system. Each segment which exists in the Archival Storage System has a Unique_ID. This Unique_ID is an integer which uniquely identifies each segment. It is chosen from a large list of integers. Since the data type is a longword, the list contains more than four billion unique integers. To prevent a communication path from existing when a segment identification must be passed out of the Kernel, an alias is provided which virtualizes the Unique_ID. When a process wishes to create a new segment, it must pass the Kernel a Mentor_Segment_# and a desired Entry_#. The mentor segment can be any segment the Supervisor wishes, but an entry for the mentor must be in the Known_Segment_Table of the process. The Segment_Manager then looks up the ASTE_# of the segment and Signals the Memory_Manager with the ASTE_# and Entry_#. The Memory_Manager maintains a flat file system known as the Alias_Table (figure 30) which is systemwide. Every active mentor segment has an ASTE_# for a segment of

Unique ID	Size	Access Class	Page Table Address	Alias Table Address

Figure 30. Alias Table

the Alias_Table. When the Memory_Manager receives a Signal which requires use of the Alias_Table, the Memory_Manager brings the appropriate Alias_Table segment into memory. The Entry_# is then used as an index into the Alias_Table where the Memory_Manager can determine the Unique_ID and physical attributes of the indexed segment. A segment exists for each entry in the Alias_Table.

The attributes found in the Alias_Table are the segment Size, the location of its secondary storage Page_Table, the segment Access_Class, and the secondary storage page table of its Alias_Table segment if it is a mentor segment. Alias_Table storage is allocated when the first request for an Alias_Table entry is made, and is deallocated whenever the segment is empty. The Memory_Manager will not honor a request to delete a segment if it has an Alias_Table segment. If this deletion were allowed, storage space would be lost forever since the Alias_Table segment of the mentor segment and any segments referred to by that Alias_Table segment would not be recovered.

d. Storage Allocation

The Memory_Manager is responsible for controlling storage media as well as main memory. The storage hardware for this design is anticipated to be a type of hard disk using the Winchester technology. However, the design may be initially implemented on an eight-inch "floppy" disk drive using the IBM standard,

single density format. Using this standard, a single disk has 77 tracks of 26 sectors each available for storage. Each sector stores 128 bytes of information.

Since the Z8000 hardware allows segment sizes in multiples of 256 bytes, it is convenient to establish a "page" size as 256 bytes. Using this scheme, a page can then be stored in two sectors of the disk. A page then becomes convenient as the size of a page table. The page table is used to record the location of each page of the segment on the disk. If the location of each page is stored in unpacked form, a total of 128 page locations can be stored in a page. Note, however, that this scheme uses only 11 of the 16 bits which can contain information (7 bits for the track index, and 4 for the sector index), and can easily be reduced to 10 bits since every other sector is not explicitly indexed. This means that 1024 pages can be addressed by one page of a Page_Table and is adequate to store the maximum size segment (256 pages) allowed by the Z8000 hardware.

A free page bit map is needed in order to record which pages on the disk are available and which are allocated. This will also require one page on the disk. This scheme allows the disk space to be allocated to segments from the "free list" and does not require complex compaction algorithms. If other forms of storage media are used they can be easily adapted to this scheme.

9. Input_Output Manager Module

The I/O_Manager is the non-distributed Kernel process which is concerned with moving information across the boundaries of the Archival Storage System. It manages the input and output ports of the system as a resource in much the same way as the Memory_Manager handled the memory resource. The I/O_Manager would use an Attach_Table to virtualize the system ports. While the I/O_Manager is a process in the general case, it can be designed and implemented as a distributed Kernel function.

III. CONCLUSION AND FOLLOW ON WORK

The detailed design of the Security Kernel for a data warehouse has been presented. This design is suitable for implementation on a Zilog Z8000 microprocessor-based system. A minimal subset of a family of secure operating systems has been demonstrated to exist and can be implemented on microprocessor hardware which is available today. This design also shows the feasibility of an Archival Storage System that can be the nucleus of a distributed, multi-microprocessor computer system by providing archival storage with multilevel security.

The design illustrates the utility of modern software engineering techniques. A loop-free structure was maintained as a design goal, preserving the ability to modify a module without introducing change in any other module. An explicit process structure simplifies the design for asynchronous functions. Functionality of this family member can be extended by including additional primitives from the larger set of primitives described by O'Connell and Richardson [5].

Security of information was a primary goal throughout the design process. A mathematical model was used as a foundation for the Kernel to insure properly designed security. A multilevel security capability is included for the storage system. Furthermore, on this base a complete, multilevel secure, distributed "system" can be constructed with the storage system as the only component requiring

multilevel security.

While designed for a single microprocessor with memory management unit support, the structure of the high level design which allows configuration independence was preserved. The same concepts for reducing bus contention in a multiprocessor system while providing data sharing were used and can be easily extended, e.g., for increased processing capacity to serve a large number of higher bandwidth hosts.

Implementation of the Archival Storage System is an area for further work. The distributed Kernel data structures and procedures are described in this thesis. Additional effort will produce compilable implementation code and from this code generate a loadable system. The Kernel non-distributed processes for I/O and physical memory management have been briefly presented and more detailed design will be needed prior to implementation. The Archival Storage System design is a minimal family member. Additional services to the Supervisor and generalization of the simplifying assumptions (e. g., to interface to multilevel hosts) are major areas where continued research is indicated.

After implementation of the storage system, substantial work is necessary in performance evaluation. Hardware choices have been primarily left to the implementor. Since many of the software design implications on efficiency are unknown at the present

time, fine-tuning of both hardware and software will result in better system performance.

```
Gate_Keeper Procedure

Type
  Parameter_Table_Entry  Record [Function_Address Longword
                                 NØ_Of_Parameters Integer
                                 Para_1_Length Integer
                                 Para_2_Length Integer
                                        .
                                        .
                                        .
                                 Para_n_Length Integer]

Local          !Initialize local variables!
  Valid := 1
  Invalid := Ø
  Index := 8

  Parameter_Table Array [Max_Function_Code
                                 Parameter_Table_Entry]
              :=[[<<Traffic_Controller>>Block_Entry,1],
                 [<<Traffic_Controller>>Wake_Up_Entry,3],
                 [<<Segment_Manager>>Create_Entry,5],
                 [<<Segment_Manager>>Delete_Entry,3],
                 [<<Segment_Manager>>Make_Known_Entry,6],
                 [<<Segment_Manager>>Terminate_Entry,2],
                 [<<Segment_Manager>>Swap_In_Entry,3],
                 [<<Segment_Manager>>Swap_Out_Entry,2]]
```

```
Entry
  DI NVI,VI         !Disable interrupts!
  PUSH @RR14,R0          !Save user registers!
  PUSH @RR14,R1
  PUSH @RR14,R2
  PUSH @RR14,R3
  PUSH @RR14,R4
  PUSH @RR14,R5
  PUSH @RR14,R6
  PUSH @RR14,R7
  PUSH @RR14,R8
  PUSH @RR14,R9
  PUSH @RR14,R10
  PUSH @RR14,R11
  PUSH @RR14,R12
  PUSH @RR14,R13
  LDCTL R2,NSPSEG          !Save user stack pointer!
  LDCTL R3,NSPOFF
  PUSH @RR14,R2
  PUSH @RR14,R3
  EI NVI,VI           !Enables interrupts!
  VALIDATE: DO         !Check location of arguments for user
                         read/write access!
            LDL <<DIST_KERNEL_ID>>ARGUMENT_POINTER,RR2
            CALL CHECK_ADDRESS_SPACE
                  !Get return value!
            LDB RH0,<<DIST_KERNEL_ID>>VALIDITY_CODE
            LDB RH1,VALID
            CPB RH1,RE0
            IF NE THEN EXIT FROM VALIDATE  !Return if invalid!
            ELSE LDL RR2,<<DIST_KERNEL_ID>>ARGUMENT_POINTER
                 LDB RE0,INDEX
            FI
```

```
MOVE_STACK: DO    !Move argument list to Kernel work space!
            CPB RH0,#0
            IF EQ THEN POP R4,@RR2
                       PUSH @RR14,R4
                       DEC RH0
            ELSE EXIT FROM MOVE_STACK
            FI
            REPEAT FROM MOVE_STACK   !Loop until all moved!
            OD
CALL FUNCTION: DO
  LD FUNCTION_CODE,@RR14(#24) !Retrieved from system call
                              instruction on system stack!
  LD R6,MAX_FUNCTION_CODE
  CP R6,FUNCTION_CODE
  IF GT THEN LD LDL RR10,<<DIST_KERNEL_ID>>MESSAGE_POINTER
              LD R2,INVALID_FUNCTION_CODE
              LD @RR10(0),R2    !Put error code into message!
              EXIT FROM CALL_FUNCTION
  ELSE LD R6,@RR2(NUMBER OF ARGUMENTS)
        !Check number of parameters!
       CP R6,FUNCTION_TABLE[FUNCTION_CODE,NO_OF_ARGUMENTS]
       IF EQ THEN CALL FUNCTION_TABLE[FUNCTION_CODE,FUNCTION]
              ELSE LDL RR10,<<DIST_KERNEL_ID>>MESSAGE_POINTER
                   LD R2,INVALID_ARGUMENT_LIST
                   LD @RR10(0),R2
                   EXIT FROM CALL_FUNCTION
              FI
   FI
   OD     !END OF CALL_FUNCTION LOOP!
```

```
    LDB RH1,INDEX      !Zero out user argument list!
    ZERO_OUT: DO
              CP RH1,#Ø
              IF NE THEN POP R2,@RR14
                         DEC RH1
              ELSE EXIT FROM ZERO_OUT
              FI
              REPEAT
              OD
    LDL RR8,<<DIST_KERNEL_ID>>MESSAGE_POINTER
    LDL RR4,@RR14(NSTACK_POINTER)
    LDB RH2,#Ø
    LDB RH1,#8
    MOVE_RET_MSG: DO    !Put message back in user area!
                  CP RH1,#Ø
                  IF NE THEN LD R2,@RR8(RH6)
                             PUSH @RR4,R2
                             INC RH2
                             DEC RH1
                  ELSE EXIT FROM MOVE_RET_MSG
                  FI
                  REPEAT
                  OD
OD    !END OF VALIDATE!
    DI NMI,VI      !Disable interrupts!
    POP R3,@RR14     !Restore user registers!
    POP R2,@RR14
    LDCTL NSPSEG,R3
    LDCTL NSPOFF,R2
    POP R13,@RR14
    POP R12,@RR14
    POP R11,@RR14
    POP R10,@RR14
    POP R9,@RR14
    POP R8,@RR14
    POP R7,@RR14
    POP R6,@RR14
    POP R5,@RR14
    POP R4,@RR14
    POP R3,@RR14
    POP R2,@RR14
    POP R1,@RR14
    POP RØ,@RR14
    EI NMI,VI      !Enable interrupts!
    IRET      !Restore pre-call cpu state!
End Gate_Keeper
```

APPENDIX B - SUCCESS AND ERROR CODES

CODE	ENTRY POINT
Invalid_Function_Code	Gate_Keeper
Invalid_Argument_Code	Gate_Keeper
Mentor_Seg_Not_Found	Create_Segment
	Delete_Segment
Not_Allowed	Create_Segment
	Delete_Segment
	Make_Known
	Wake_Up
Not_Compatible	Create_Segment
Segment_Too_Large	Create_Segment
No_Segment_#_Avail	Make_Known
Segment_Found	Make_Known
	Swap_In
	Swap_Out
Segment_Not_Known	Terminate
Segment_In_Core	Terminate
Kernel_Segment	Terminate
Invalid_Segment_#	Terminate
Swapped_In	Swap_In
Swapped_Out	Swap_Out
Queue_Empty	Block
Queue_Overflow	Wake_Up
Inserted	Wake_Up

CODE	ENTRY POINT
Not_Related	Non_Disc_Security
Greater_Than	Non_Disc_Security
Less_Than	Non_Disc_Security
Equal	Non_Disc_Security

LIST OF REFERENCES

1. Schroeder, M. D., Clark, D. D., and Saltzer, J. H., The Multics Kernel Design Project, paper presented at ACM Symposium on Operating System Principles, 6th, November 1977.

2. Mitre Corporation Report 2934, The Design and Specification of a Security Kernel for the PDP-11/45, by W. L. Schiller, May 1975.

3. Parks, E. J., A Design of a Secure, Multilevel, Multiprogrammed File Storage System for a Microprocessor Environment, MS Thesis (in preparation), Naval Postgraduate School, 1979.

4. Smith, D. L., Method to Evaluate Microcomputers for Non-Tactical Shipboard Use, MS Thesis, Naval Postgraduate School, September 1979.

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

6. Schell, Lt.Col. R. R., "Computer Security: The Achilles' Heel of the Electronic Air Force?," Air University Review, v. XXX no. 2, January 1979.

7. Schroeder, M. D., "A Hardware Architecture for Implementing Protection Rings," Communications of the ACM, v. 15 no. 3, p. 157-170, March 1972.

8. Lampson, B. W., "A Note on the Confinement Problem," Communications of the ACM, v. 16 no. 10, p. 613-615, October 1973.

9. Lipner, S. B., "A Comment on the Confinement Property," Operating System Review, v. 9, p. 192-195, November 1975.

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

11. Madnick, S. E. and Donovan, J. J., Operating Systems, McGraw Hill, 1974.

12. Denning, D. E., "A Lattice Model of Secure Information Flow," Communications of the ACM, v. 19, p. 236-242, May 1976.

13. Peuto, B. L., "Architecture of a New Microprocessor," Computer, v. 12 no. 2, p. 10, February 1979.

14. Snook, T., and others, Report on the Programming Language PLZ/SYS, Springer-Virlag, 1978.

15. Zilog, Inc., Z8000 PLZ/ASM Assembly Language Programming Manual, 03-3055-01, Revision A, April 1979.

16. Zilog, Inc., Z8001 CPU Z8002 CPU, Preliminary Product Specification, March 1979.

17. Zilog, Inc., An Introduction to the Z8010 MMU Memory Management Unit, Tutorial Information, August 1979.

18. Dijkstra, E. W., "The Structure of 'THE' Multiprogramming System," Communications of the ACM, v. 11 no. 5, p. 341-346, May 1968.

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

20. Millen, J. K., "Security Kernel Validation in Practice," Communications of the ACM, v. 19 no. 5, p. 243-250, May 1976.

21. Knuth, D. E., The Art of Computer Programming: Volume 1/Fundamental Algorithms, Addison-Wesley, 1968.

THE DESIGN OF A SECURE FILE STORAGE SYSTEM

by

Edward James Parks
Lieutenant, United States Navy
BS, United States Naval Academy, 1971

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1979

Dean of Information and Policy Sciences

ABSTRACT

A design for a secure, multi-user, File Storage System is developed. This design, incorporating a concurrently developed Security Kernel, provides a multilevel secure flexible file storage serving a distributed system of dissimilar computers. The Security Kernel is responsible for non-discretionary (e.g., classification and clearance) security and the File Storage System Supervisor is resporsible for discretionary (e.g., "need to know") security. Multilevel security is achieved by the controlled access to consolidated file storage for Host computer systems. Multiprogramming of surrogate Supervisor processes operating on behalf of the Host computer systems provides for system efficiency. A segmented memory at the Supervisor level allows controlled data sharing among authorized users. System integrity is independent of the internal security controls (or lack of them) in the distributed systems; the File Storage System prevents system-wide security side effects. A loop free structure along with system simplicity and robustness are design characteristics.

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.

Special thanks go to Lt. Col. Roger Schell and Prof. Lyle Cox for their invaluable advice and many hours of assistance.

I. INTRODUCTION

Lack of data security is a central issue in computer science today. Data security can be divided into external physical aspects (i.e., guards, fences, etc.) and internal system aspects (i.e., internal software and hardware operations); both of which are necessary for effective system security. The physical aspect is understood and does not pose a significant problem today. Continued loses (viz., money, data) due to computer 'error', illustrate that the second aspect of data security, viz., internal security, has not been solved and continues to be a problem.

This shortcoming results from the fact that internal computer security has not been a mandatory design objective during hardware and/or software selection and/or production in most (if not all) contemporary computer systems. This renders them prone to security violations from accidential or malicious penetrations [Schell(1)]. Ad hoc attempts to provide the necessary system security in the later stages of the system design or implementation have not generally met with success.

In contrast, this thesis presents a design for a multilevel secure computer operating system, the File Storage System (FSS) in which internal computer security is a primary design objective. There are two goals this system is designed to achieve: 1) to provide sharing of data among authorized users and, 2) control access to a consolidated "warehouse" of data. This controlled access to consolidated

data, predicates a "star" network for the system structure as depicted in figure 1. It must be noted, however, that the FSS cannot control the physical security of the Host systems and that Host systems have the ability to circumvent FSS security by direct inter-Host communication links. To preserve data security, all accesses to the FSS consolidated data must go through the FSS for access validation.

Data sharing among authorized users is accomplished by a segmented environment which allows controlled direct access to all on-line data. The Security Kernel (or simply Kernel) is used to insure that non-discretionary data access is performed in an absolutely controlled (i.e., secure) manner. (See [Coleman] for detailed information on the Security Kernel.)

A. PROBLEM DEFINITION

"It is illogical to ignore the fact that computers may disseminate information to anyone who knows how to ask for it, completely bypassing the expensive controls placed on paper circulation." [Schell(1)]

That this fact is ignored is demonstrated by the estimated 100 million dollars lost yearly by non-secure computer systems in the United States [Denning(2)]. It is obvious that a primary problem/limitation of computer systems in use today is the lack of data security. As requirements to store and access data by computer increase, the seriousness of this problem/limitation cannot be ignored.

A system that can simultaneously provide data at

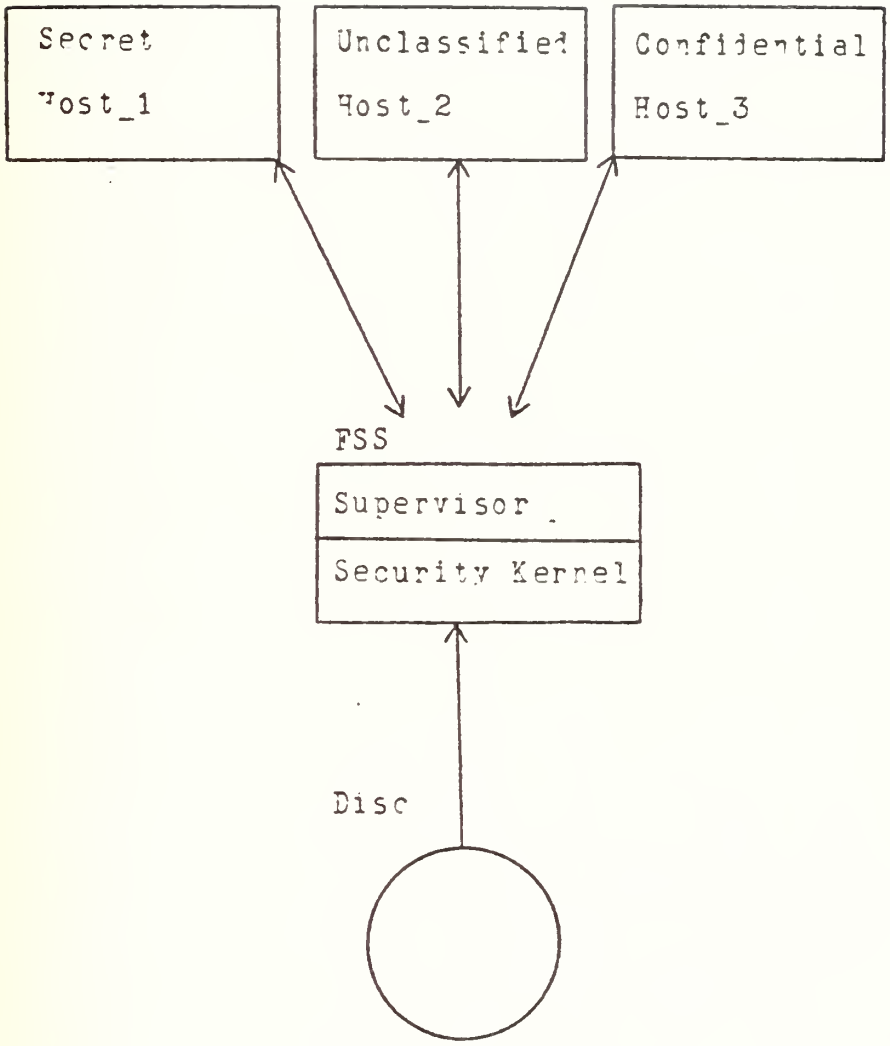


Figure 1. System Configuration

different sensitivity (viz., "classification") levels for users with different access authorizations (viz., "clearances") without a security violation is said to be a multilevel secure system. Because it is usually not desirable to authorize all system users access to the highest level of data ('system high') or provide separate (without sharing) systems for each level of data, a multilevel system is highly desirable. A multilevel system also allows the maximum amount of controlled data sharing among authorized users, a primary goal of any data storage system.

Previous research shows that a viable approach to the question of internal computer security exists. This approach, sometimes termed the "security kernel approach" [Schell(2)], was introduced by Schell in 1972. It gathers into one module all elements that effect the system security. The module, by being restricted in size, can be verified correct which in turn allows the total system to be certifed secure.

The FSS software is composed of the Supervisor and the Kernel. It will provide a multilevel secure consolidated file storage for distributed Host computer systems. The non-discretionary security provided by the Kernel and the discretionary security provided by the Supervisor will implement a wide range of security policies, including the standard Department of Defense (DOD) security policies. Data sharing is achieved by a segmented memory environment at the

Supervisor level. The Supervisor uses segments (invisible to the Host systems) to construct the Host files. Multilevel security is achieved by the management of files submitted by the Host systems which exist at distinct security levels. This allows the construction of a multilevel secure system which is dependent on only one secure element of the FSS--the Kernel.

B. BACKGROUND

The dramatic reduction in size and cost along with the increase in performance of microprocessors in the last decade has made their use feasible in areas that have previously been reserved for mini/maxi computers (or not computed at all). Whereas security has been notoriously lacking in the larger systems, it has been non-existent in microprocessors to date.

Because of their small size, low cost, durability, and, perhaps most importantly, the manpower savings induced (just to mention a few of many advantages), microprocessors have high appeal for use in a military environment. However, the military also has an obvious need for security within their computer systems, whether they are micro, mini, or maxi based.

For example, the Navy is presently considering systems for the next generation of non-tactical shipboard computers [Smith]. They will be mainly used for data processing in the areas of:

Pay and Personnel

Supply and Finance

Maintenance.

Cost, size and speed constraints will soon be met by commercially available products. Security, however, continues to be a problem not adequately addressed in any available systems. To preserve data confidentiality (not only with respect to clearance level but also with respect to the current stipulations of the Privacy Act), security is a necessary part of any shipboard computer system. Pay records, for example, should not have the same access level as maintenance records. In order to store records in a common data base and to have controlled sharing when appropriate, the computer must be able to maintain a multilevel secure environment.

There are several possible approaches to achieve a secure multilevel environment. The frontal approach, which is most difficult, is to certify all distributed computers which have access to the data base as secure. A second method and the method adopted for the FSS, is to cerfify only one element of the FSS secure--the Security Kernel. All access to the FSS that involves non-discretionary security will be validated by the Kernel. The FSS therefore guarantees to manage files in a manner consistant with the FSS security policies.

The design for the FSS is one member of a family of systems proposed by O'Connell and Richardson [O'Connell].

Security, configuration independence, and a loop-free structure are characteristics of this family of systems.

C. BASIC DEFINITIONS

1. Security

Although any viable secure system includes both internal and external aspects, relying excessively on external controls is not desirable in many cases due to the added expenses and increased security risks involved in error-prone manual procedures. External controls also cannot provide the secure sharing of data that is needed in such applications as integrated data bases and computer networks, primary characteristics of the FSS. The use of the Kernel concept is a demonstratively effective and practical method for providing the internal computer security controls that are necessary for a secure multilevel system. This concept is at the center of the FSS design.

The basic concept behind this approach is that a small portion of hardware/software, the Kernel, can provide the internal security controls that are effective against all attacks, (malicious or accidental) including those never thought of by the designer. (This also means that errors in the FSS Supervisor cannot cause unauthorized access to data.)

System security is the implementation of a security policy. This policy is a collection of laws, rules, and regulations that establish the rules for access to the data

in the system. Such policies, such as the one established by the DOD, have two distinct aspects: discretionary and non-discretionary security. Non-discretionary security externally constrains what access is possible. In the DOD environment, the familiar non-discretionary security levels are: top secret, secret, confidential, and unclassified. Since most contemporary computer systems do not provide the data labeling necessary to support non-discretionary security, all data is implicitly accessible. In the FSS, segmentation allows unique identification and labeling of data; non-discretionary security is therefore supported. The Kernel is the one element in the FSS responsible for enforcing non-discretionary security.

Non-discretionary security involves the comparing of the access class of a specific object (object access class, (oac)) with the access class of the requestor (subject access class, (sac)) to insure compatibility. In a DOD environment, for example, a person (subject) with sac of secret has access to files (objects) at any access class equal to or less than secret. The relationships between different access classes are represented by a partially ordered lattice structure [Denning(1)]. This lattice represents the authorized access based on the relationships of two levels. An example of the not-related (making the lattice partially ordered) relationship, occurs because of DOD compartmentalization (e.g., secret is not related to secret.nuclear). The following accesses are permitted for

the relationships represented by this lattice structure.

```
sac  =  oac  :read/write access
sac  >  oac  :read access (read down)
sac  <  oac  :write access (write up)
sac  <> oac  :no access (sac not related to oac)
```

In each case, the Kernel must know the identification of the Host system if it is to perform correct non-discretionary security checks. Unique system identification is provided by the system port number, which is hardwired, and known to the Kernel.

Discretionary security provides a refinement to the non-discretionary security policy and is reflected in the DOD "need to know" policy. Computer systems which have Access Control Lists (ACL) associated with data, implement this discretionary policy. The FSS Supervisor is responsible for the System discretionary security and although this aspect of the System security is not validated by the Kernel (and therefore not certified correct), the validity of the non-discretionary security is not affected.

To implement its aspect of security, the Supervisor needs to know the identification of the Host system "user". This Host system user identification must be passed to the FSS Supervisor by the Host system. Since an insecure Host system cannot be trusted to pass the correct information, the user identification is only as good as the Host system implementation. (i.e., FSS discretionary security is only as

good as the Host System's implementation of discretionary security.) This implementation may be good on some systems, (e.g., UNIX [Morris]) but non-existent on other systems (e.g., CP/M [Digital]). It must be remembered that this in no way affects the enforcement of the non-discretionary security by the Kernel.

2. Process

A process can be described as a locus of execution. The collection of locations that may be accessed during this execution is known as the process' address space [Madnick]. A process also has the characteristic that it may be executed in parallel with other processes, enhancing system efficiency and allowing the separation of tasks into different processes for design clarity.

The FSS has two processes per Host system. These are an input/output (IO) process for Supervisor to Host data transfer and communication and a file management (FM) process that controls and maintains the Supervisor file structure. Interprocess communication is achieved by the use of eventcounts, sequencers, and synchronization primitives internal to the Kernel (described later).

3. Segmentation

Segmentation allows for the direct addressing of all system on-line information and the application of access control to this information. Note that direct addressing

does not mean random access to the on-line information. On the contrary, access to segments is controlled by explicit memory management calls to the Kernel to swap in/out a segment. A segment can be defined as a logical grouping of information such as a subroutine, procedure, data area, or file. Each processes' address space consists of a collection of segments. In a segmented environment, all address space references require two components, a segment specifier and an offset within that segment. Segmentation is used to provide the Supervisor domain of each process a virtual memory of limited size. Segments, as mentioned earlier, are used by the Supervisor to construct the Host files which retain the attributes of segments (i.e., access control).

4. Multiprogramming

A multiprogrammed environment is one in which more than one process is in a state of execution at the same time. These processes share processor time, memory, and other resources among the active processes. In the design for the FSS, the Supervisor processes are multiprogrammed in an asynchronous manner for system efficiency. A multiprogramming environment allows the Host systems to operate in a logically parallel manner which adds to System design simplicity and clarity.

5. Protection Domains

One of the key elements necessary for valid Kernel

implementation is the isolation of the Kernel from all possible outside influences. This can be done through the use of protection domains.

Protection domains are used to arrange process address spaces into "rings" [Schroeder] of different privilege. This arrangement is a hierarchical structure with the most privileged domain being the inner most ring. Figure 2 represents the ring organization in the FSS.

Protection rings may be created by either hardware or software. Hardware is more efficient but is not commercially available in microprocessor devices today. Two state devices are available, however, and by implementing the two states as separate rings and providing for software ring crossing mechanisims, the necessary two protection rings can be created.

D. SYSTEM REQUIREMENTS

There are no fixed hardware requirements for the implementation of the FSS. System efficiency does, however, depend on an appropriate choice of hardware. Two basic hardware features that are felt to be necessary for a viable implementation of the FSS are segmentation and multiple domains.

Segmentation is necessary for access control and data sharing. A multiple state (two in this case) is necessary for the isolation of the Kernel from the remaining (and uncertified) software.

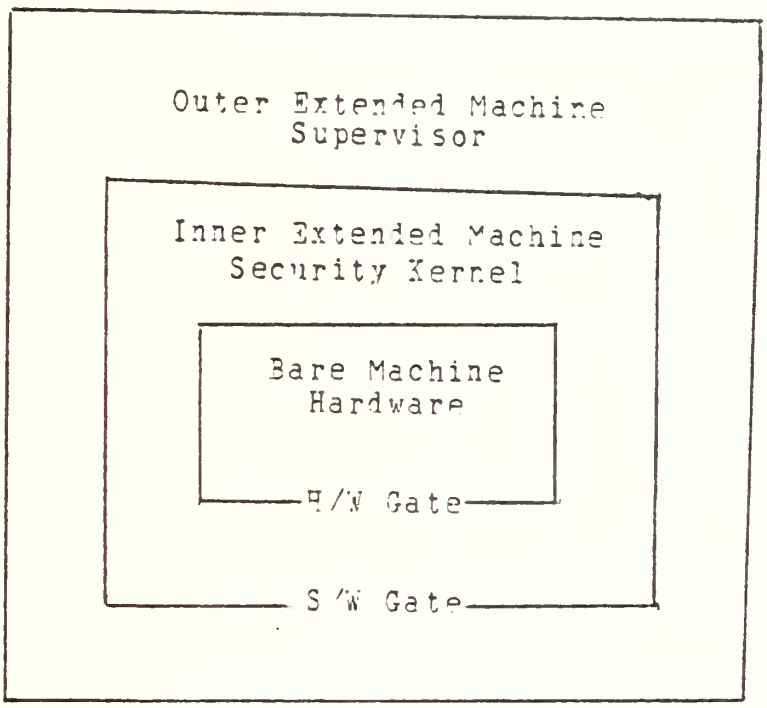


Figure 2. Protection Domains

Only the Kernel access to privileged machine instructions and [illegible] all system input/output. It provides a segmented [illegible]ent in which the Supervisor operates. The Super[illegible] turn, provides a virtual file environment for the [illegible]ter systems.

II. DESIGN

A. HARDWARE SELECTION

A secure computer system is not dependent on the hardware on which it is implemented. However, as mentioned above, segmentation and multiple domains are considered necessary for FSS efficiency.

Segmentation allows the use of one uniform type of information object, the segment, at the Kernel level. This simplifies Kernel design and contributes to keeping Kernel size small. A segment address consists of a segment name and offset within the segment. Although this addressing can be done in software, it is faster and more efficient when done in hardware. Hardware can also simultaneously check for authorized access, a necessary feature of a secure system.

Multiple domains are currently used in some of the larger machines to protect the operating systems from the applications programs. Multiple domains have not, until recently, been available in a microprocessor configuration. The FSS design requires only two domains, one for the Kernel and one for the Supervisor.

The introduction of the Zilog Z8000 series microprocessor meets both the segmentation and multiple domain requirements. The FSS is targeted for implementation on the Z8001 segmented microprocessor [Zilog(2)] with its associated Memory Management Unit (MMU) [Zilog(1)]. The Z8001 is a 16 bit two-domain machine which produces a 23 bit

logical address. The Z8010 MMU maps the 23 bit logical address into a 24 bit absolute address and allows the capability of addressing up to 128 segments (with two MMU's) of 64K bytes each (8M-bytes total) in a two-dimensional memory space. (See [Coleman] for further details.) RS-232 bus compatibility is assumed for serial data input/output at the hardware level. This allows byte synchronization and byte parity checks to be performed at the hardware level by the FSS universal asynchronous receiver-transmitter (UART).

B. SYSTEM STRUCTURE

1. System Levels

Abstraction is a way of avoiding complexity and a mental tool for approaching complex problems [Dijkstra(2)]. The use of abstaction allows the presentation of a system design that is concise, precise, and easy to understand. There are four levels of abstraction for the FSS as presented in figure 3.

Level 0 is the hardware level and consists of the Z8001 microprocessor memory and some form of disc storage (initial implementation may be with floppy disc).

Level 1, the Kernel, is isolated and protected from manipulation (accidential or malicious) by being placed in the more privileged domain of the Z8001. Only the Kernel has access to "system" machine instructions and controls all access to the system hardware elements (memory, disc). The Kernel provides a segmented environment in which the

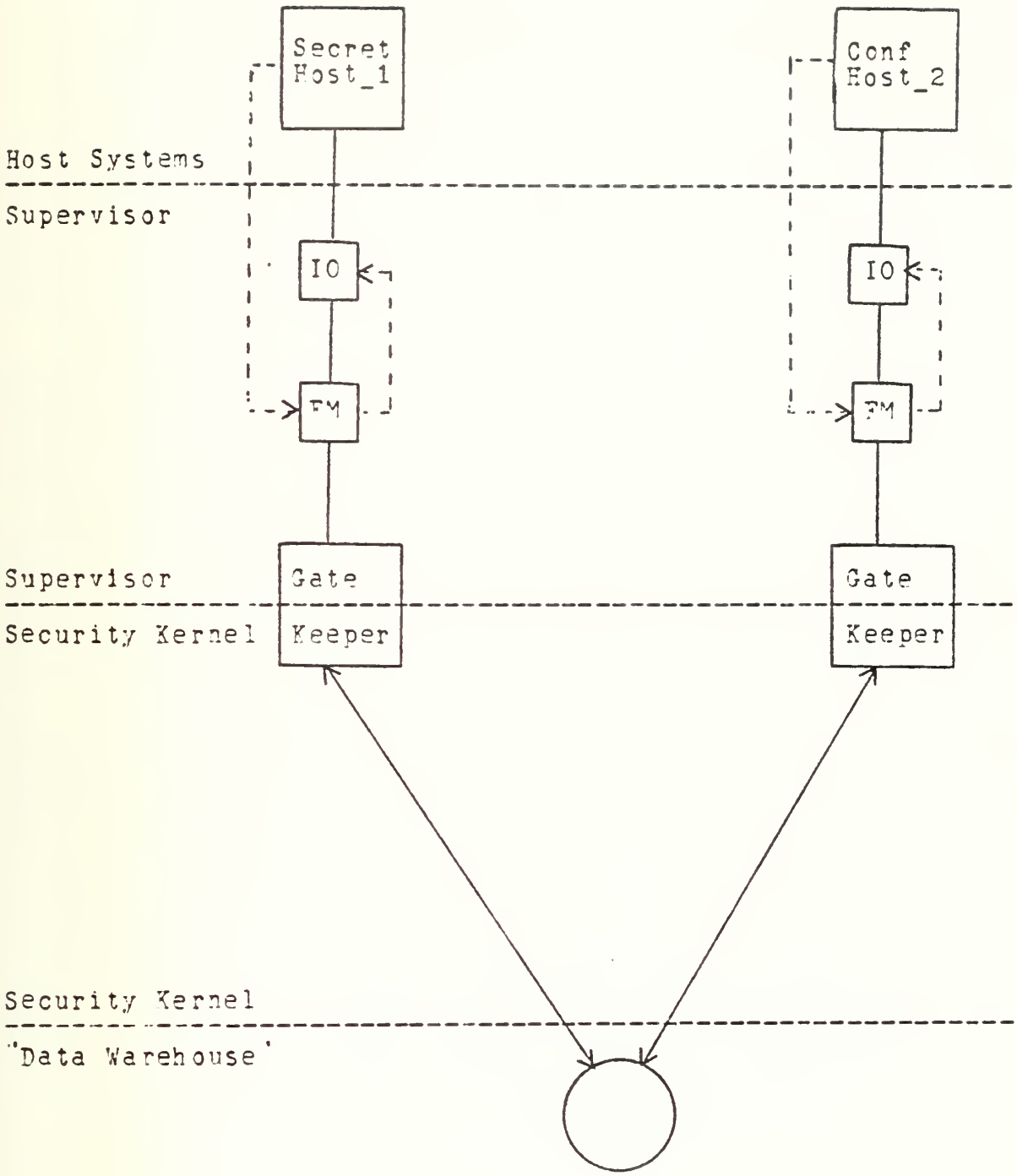


Figure 3. Abstract System View

Supervisor operates.

Level 2, the Supervisor, operates in the outer (less privileged) domain of the Z8001. It has access to "normal" machine instructions, but must go through the software Gatekeeper [Coleman] of the Kernel to get access to memory (viz., segments) and disc storage. The Supervisor provides a virtual file hierarchy to each Host system for file storage. In order to manage the file hierarchy, surrogate processes (input/output (IO) and file management (FM)) are assigned to each Host system. These processes act on the requests submitted by the Host computer systems. All processes are created at system generation time and are not created or deleted in a dynamic manner.

Level 3 consists of the Host computer systems. These systems are hardwired to the Z8001 in the FSS design. Each port has a fixed access level so that if a multilevel secure Host desires to handle data at two levels, it must have two connections to the FSS. (Note that if the Host is not a true secure multilevel Host, and does have multiple connections with distinct levels, then the FSS security constraints are circumvented.)

2. System Protocol

Protocols are formal specifications which constrain data exchange between systems and the FSS. These specifications allow the FSS to achieve bounded, deadlock free and fault tolerant communication. To organize and

simplify protocol design in the FSS, protocol is logically divided into a hierarchical structure of two interacting layers. Level 1 protocol handles packet (descrited later) synchronization, error detection, and command type determination. Level 2 handles the repetitive activity of data transfer.

Data and commands are transmitted between FSS and Host via fixed size packets. Packet synchronization is necessary for Host-FSS communication. Error detection/correction is closely related to the problem of packet synchronization; packets not in synchronization will not be correct. The converse is not true, however. A synchronized packet may contain transmission errors. There are several methods for error detection/correction [Hamming]. A design choice of a simple check sum per packet (to detect packet errors) was made in the interest of System simplicity. If an error is detected in a packet, the Host will be requested to stop packet transmission and to begin again with the packet in which the error was detected. Of course, the FSS must be able to provide the same service. This retransmission upon error detection strategy, combined with the byte parity checks performed at the hardware level by the UART, will provide the error detection/correction scheme in the initial FSS design.

3. Host Environment

The job of the FSS is to provide a service, viz., to

store files in a secure 'data warehouse'. The files are submitted by various Host computer systems. The virtual environment provided the Host systems is therefore a primary design consideration of the overall FSS design. Design goals are to make this Host environment simple, easy to use and understand, efficient and robust.

The center of the Host environment is the hierarchical file structure maintained by the Supervisor of the FSS. This file structure is a tree organization which facilitates design abstraction (virtual file systems per Host) as well as file system searches via tree traversal. Figure 4 illustrates the overall logical structure of the Supervisor file system.

A file can be defined, in the case of the FSS, as one or more Supervisor segments grouped together for the purpose of access control (security), retrieval (read), and modification (write) [Shaw]. In the FSS the file is the basic unit of storage at the Host system level.

The hierarchical file system contains two types of files: 1) data files, and 2) directory files. Both file types are constructed from segments (invisible to the Host systems) at the Supervisor level. The characteristics usually associated with a segmented environment (Supervisor level) such as data sharing and access control, are transferred to the file environment (Host level) by the FSS.

The Host system environment consists of a virtual file hierarchy maintained for each Host system (i.e., one

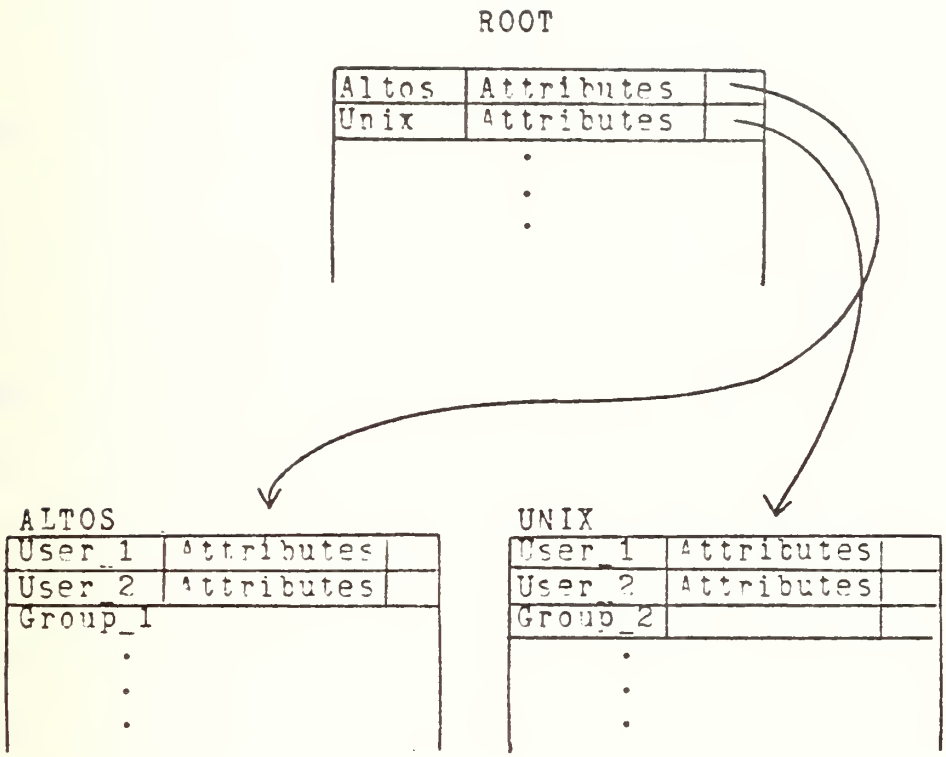


Figure 4. General Supervisor File Hierarchy Example

virtual file system per hardware port). A primary reason for having multiple virtual file hierarchies is to avoid the problem of naming conficts which would eventually occur in the Supervisor hierarchy as the system grew if per-host virtual file systems did not exist. Multiple directories also allow the Host systems to group related files into one directory, simplifying search and Host use. The Supervisor will control the duplication problem within a virtual file system by not allowing duplicate file names in a single directory file. Pathnames are required to uniquely identify files in the Supervisor file systems and must be included in the Host request.

Access to the Supervisor file hierarchy is controlled in both a discretionary and non-discretionary manner. The non-discretionary access is controlled by the Kernel which will prevent a Host system from reading up or writing down (confinement property). Discretionary access to the files is handled by the Supervisor which compares the Host.user (Host user combination) with the file ACL. Requested access is permitted only if the Host.user is explicitly permitted access by the file ACL.

Each Host system virtual file hierarchy is constructed from data files and directory files which, as mentioned above, are constructed of Supervisor segments. Although dynamic growth and shrinkage are usual segment attributes, a design choice for System simplification was made to fix segment size at three increments, SMALL (512

bytes), MEDIUM (2K bytes), and LARGE (8K bytes). These sizes were chosen as a compromise between expected file sizes, Supervisor buffer requirements, and minimizing the number of software ring crossings that would be required during a data file 'read' or 'store' operation. Because segment size is limited and there exists the likelihood of encountering files larger than the maximum segment size, the concept of a multiple segment file (msf) is known to the Supervisor.

Figure 5 depicts the general tree structure of a Supervisor virtual file hierarchy. Directory files are represented by squares and data files by circles. Data files, as their name implies, contain data only. Directory files are constructed of a header and zero or more "entries". There are two types of entries, branch entries and link entries.

Branch entries contain the attributes of the file which they identify. In figure 5, for example, the attributes of directory file User_1 (entry name, ACL, size, type, etc.) are contained in directory file Host_1, branch entry User_1. One branch entry designates one Supervisor segment.

A link entry, represented by the dotted line in figure 5, is composed of an "entry name" (link name) and a pathname. (A pathname is the concatenation of entry names starting from the root directory and proceeding in sequential order to the specified file.) Like a branch entry, a link entry is used to access a specific file. For

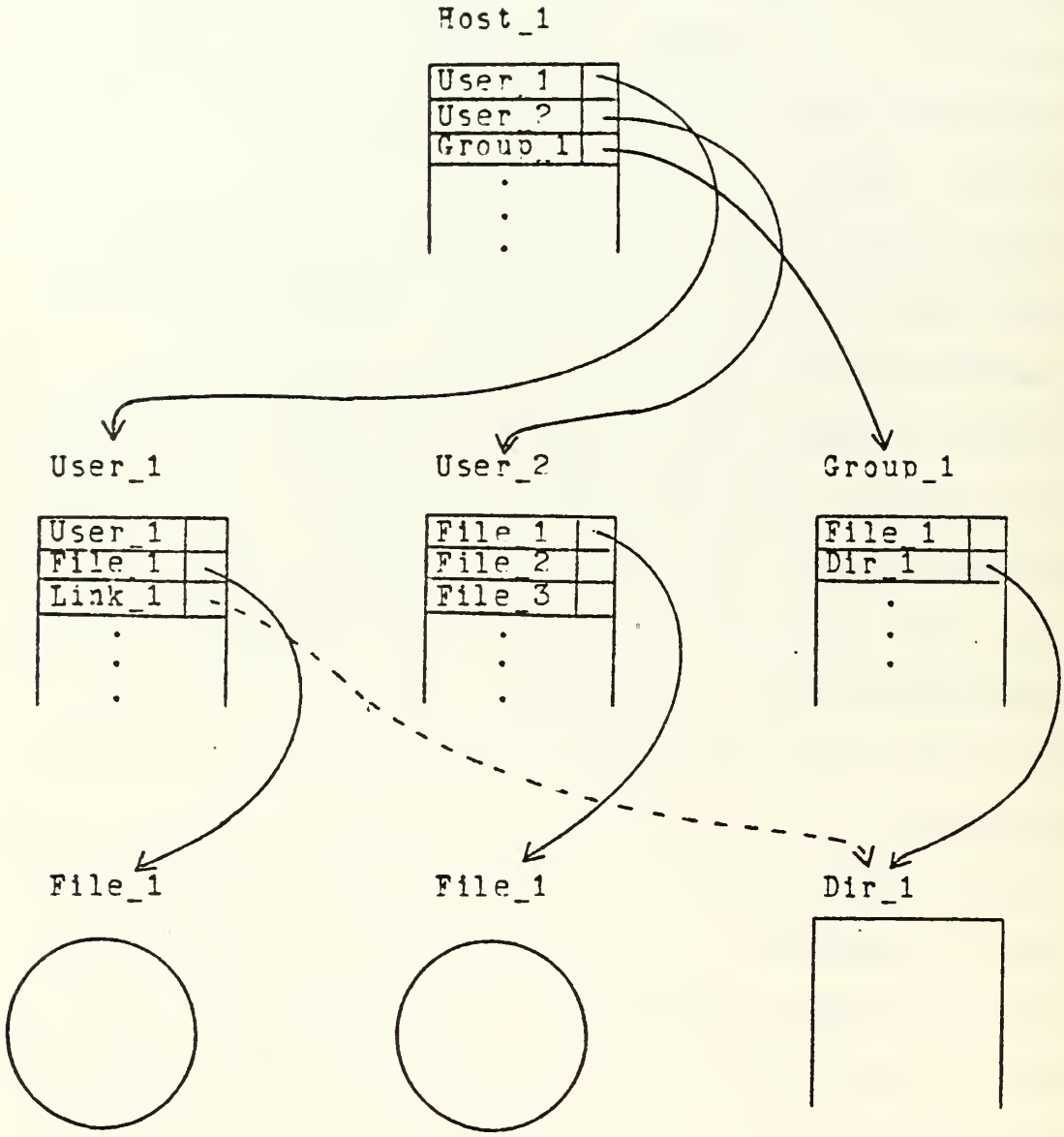


Figure 5. Virtual File Hierarchy (logical view)

example, in figure 5, the pathname contained in the link entry is Host_1>User_3>Dir_1. Unlike a branch entry, however, the link entry does not contain any file attributes. Access is controlled as the Supervisor traverses the specified path to the requested file.

The use of link entries allows sharing of files among Host systems and among Host system users. Loops which might be generated by two links which reference each other, are prevented by the Supervisor. (Loops could present a tree traversal problem to the Supervisor.)

Each file has a file name (Entry_Name--unique per directory file) given by the Host system at file creation time. This file name and its pathname are used to uniquely locate the file in the Host's virtual file system. By traversing the virtual hierarchy, the Supervisor can locate the requested file if it exists in the system. In either case (viz., whether the file exists or not), appropriate action can be taken by the Supervisor.

a. Directory File

Figure 6 is a logical representation of a file directory. Each directory file is made up of a header and zero or more fixed size branch/link entries. A fixed directory size of LARGE (8K bytes) was chosen to insure a reasonalble amount of directory space for Host system use. This could pose a "space" problem, especially for secondary storage. (Adequate main memory can be installed for required

Directory File

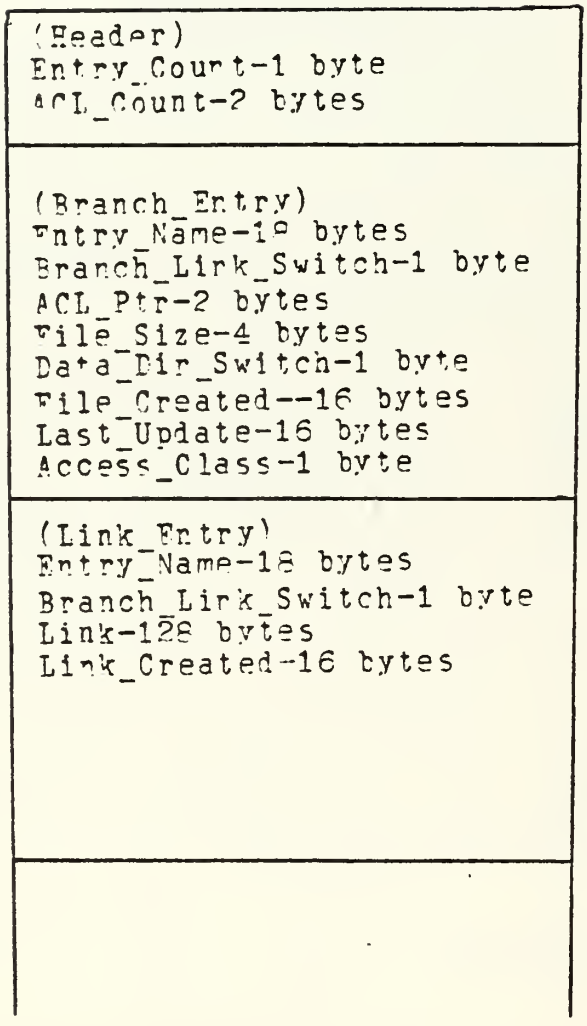


Figure 6. Logical Directory Structure

buffer space.) The Kernel, which stores segments as pages, may want to 'compact' segments by not storing on secondary storage pages which contain all "zeros". This would greatly reduce the amount of wasted space on secondary storage. (Another equally viable solution, but not selected for this design, is to have multiple segment directories in the Supervisor similar to multiple segment data files.) The directory file header contains the following information:

Entry_Count: This is the number of branch/link entries in the directory.

ACL_Count: This is a count of the number of ACL_ENTRY elements left in a "pool" of such elements.

If the entry is a branch entry, it will contain the following elements:

Entry_Name: Entry name is the file name. The Host systems are resporsible for supplying these names but, as mentioned above, will be prevented by the Supevisor from having duplicate names (file names) in one directory file.

Access_Class: This element contains the file access level.

Branch_Link_Switch: This element will identify the entry as a branch entry which in turn specifies the entry format.

ACL_Ptr: This element will point to an ACL for the branch entry. The FSS has only three distinct discretionary access modes: 1) "null" access as the name implies, declares that no access is to be allowed to the

specified Host.user combination, 2) "read" access allows a qualified Host.user to read a file only (i.e., no write access), 3) "write" access allows a Host.user write access to a file (also implicit read access). The actual ACL will be a list of authorized users in the form Host.user with an associated access mode. A 'don't care' authorization (in this case a *), will allow general access in that category. For example, *.user would allow the specified 'user' to access this file from any connected Host system with a specified access mode. This ACL for entry "user" can easily be expanded to include other categories such as "project" to further refine the discretionary access allowed to a file.

File_Size: This information is necessary for proper management of the Host READ_FILE and STORE_FILE commands by the Supervisor, viz., it allows the Supervisor to calculate the number of segments that make up a multiple segment file. It will be supplied by the Host system in the STORE_FILE command request (in bits).

Data_Dir_Switch: This switch tells the Supervisor the type of file to which the branch points (data, directory). This is necessary due to the different file formats.

File_Created: This element is used for general audit purposes. i.e., to have a permanent record of the file creator and the time of creation.

Last_Update: This element will identify the last Host and user to store into the file. This identification

will be of the form Host.user.date.time. This will allow the FSS to have a limited audit capability. The confinement property prevents the FSS from also keeping track of read accesses since processes at higher levels can read at lower levels but cannot write the audit information. Also note, that the Last_Update information for upgraded directories may not be accurate for the same reason.

If the entry is a link entry, it contains only four elements. These are: 1) Entry_Name to identify the file, 2) Branch_Link_Switch to identify the entry type, 3) Link, a pathname to uniquely identify a file, and 4) Create_Time, the time of link initiation along with the Host.user who created the link. All attribute checking is done as the Supervisor traverses the specified path.

A FSS design choice is to limit all pathlengths to 128 bytes. This places some restrictions on the Host in that long file names will soon consume the bytes available for a pathname. However, this restriction can be overcome by pathnames which contain several link entries, which can themselves be 128 bytes. With 32 branch/link entries per directory, there are an average of 32 ACL entries (3 bytes each) available to each branch entry. (Remember, link entries do not have ACL entries.) Figure 6 contains the initial field sizes for the directory construction. The primary factor in calculating the size of branch/link entries is the size of the link pathname. This increases the size of link entries to 163 bytes and although space is wasted in branch

entries, the simplification of System design resulting from a fixed size of branch/link entry is felt to be sufficient justification in the initial design.

b. Data Files

Data files are always "leaf" nodes in the file hierarchy and contain only data.

c. Multiple Segment File Directory

A msf directory is a Supervisor construct (invisible to Host systems) to manage files larger than the maximum fixed segment size. Because the number of segments that will be required by the Supervisor to store a file can be calculated from the file size information passed by the Host, a msf directory need only be a segment of size zero. This makes the Kernel alias table (which is a fixed size--see [Colemanl) the limiting factor in the maximum file size. The alias table has the same number of entries as a Supervisor directory (viz., 32) which limits maximum Host file size to 256K bytes. Files that exceed the maximum file size must be split by the Host system. An attempt to store a file that is 'too' large will result is an error condition response to the Host and an unexecuted command.

4. Host System Commands

The Host commands provide the only interface that a Host system has with the FSS. Each command is interpreted by

the FSS and acted upon by surrogate Supervisor processes; the Host system has no direct access to the FSS. There is one acknowledgement between the Host and FSS at this level. This is a "command complete" acknowledgement that informs the Host system that the Supervisor has completed action on its request. If an error condition occurs, the appropriate error code is returned in the acknowledgement.

Another aspect of the Host environment needs to be defined also. The Host environment can be divided into two states; they are the "old" state, before the FSS has acted upon the Host request, and the "new" state, which occurs after action has been completed by the FSS. The specific state of the FSS at any instant is indeterminate at the Host level if more than one Host is accessing the same file of the FSS at one time. That is, since Supervisor processes execute in a completely asynchronous manner, the FSS state may change after a Host command is sent but before the FSS acts on the command. This will not affect the performance of the System or validity of its security; Host commands will be executed as a single, atomic operation in the FSS state in which they are received and interpreted. The Host will get some "correct" response for some state existing between the sending of the Host command and the FSS acknowledgement on the same command. This allows several Hosts to safely synchronize their actions external to the FSS.

The following is considered to be a minimal subset of commands available to the Host System for adequate file

control. Figure 7 illustrates the required discretionary access attributes. The files are referenced in the Host command descriptions starting from the root of the Host virtual file system. The pathname specifies the parent directory file (containing access attributes of the file), and the file (data or directory) to which the Host command refers. All commands require a pathname for unique file identification. Each command also requires the specification of the Host system "user" in order for the Supervisor to perform discretionary security checks. This 'userid' will be supplied by the Host system or the Host system user, which ever is appropriate.

CREATE_FILE <pathname, access_class, file_type (directory, data)>. This command requests that the Supervisor create a branch entry in the specified directory under the specified file name at the specified access class. An initial access mode of write will be given to file creator and may be altered by the use of the ADD_ACL_ENTRY and DELETE_ACL_ENTRY commands. This is the only Host command where file access class is specified. It is used in this command to create upgraded directory files, if desired. (Data files may not be upgraded--described later.) In the initial implementation (with single level Hosts), there will be no upgraded directories within a Host virtual file system. Initial data file size is zero; initial directory file size is LARGE (8K bytes). Actions taken:

1) The Supervisor locates the root of the virtual

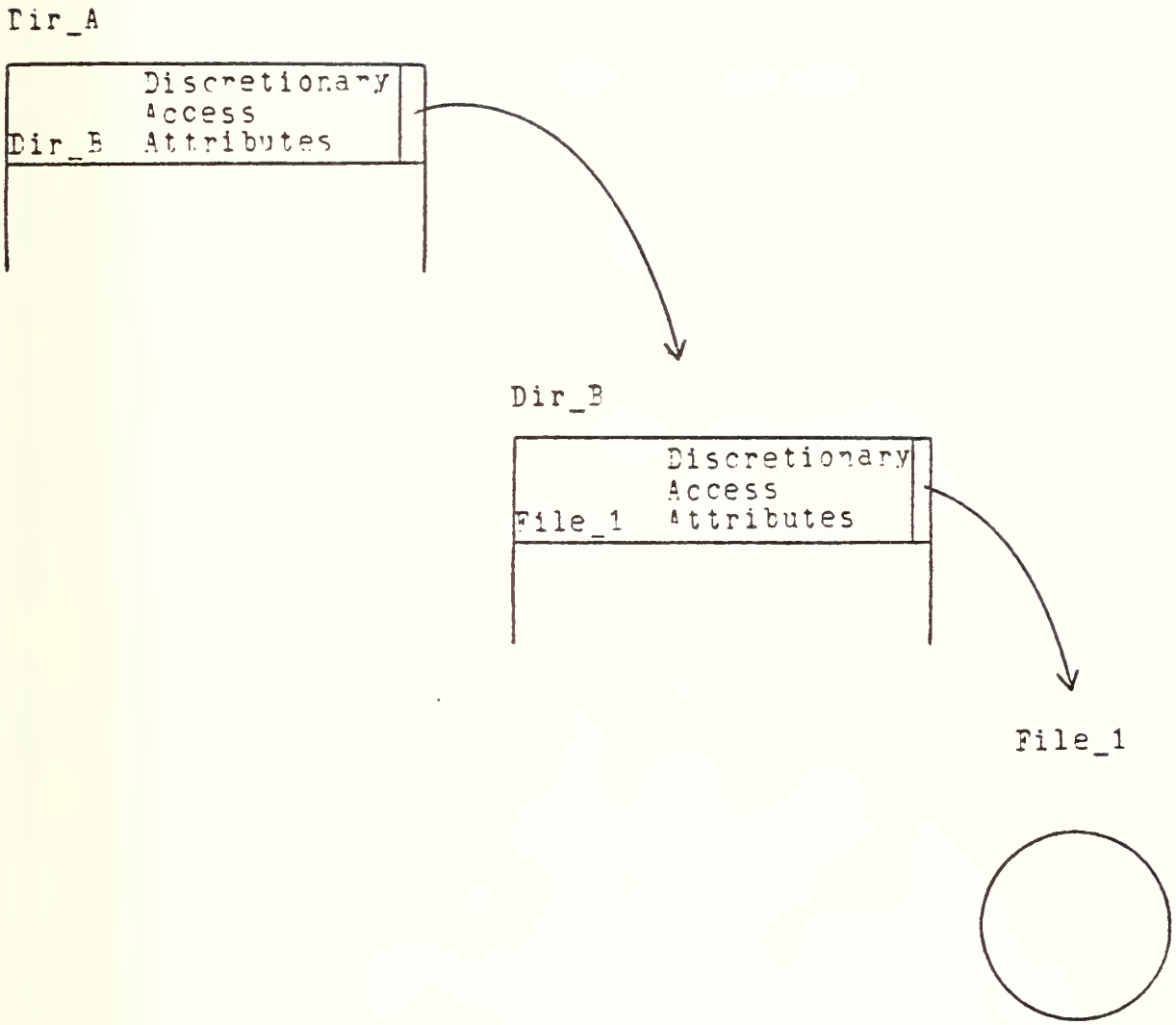


Figure 7. File Discretionary Access Control

file system for this Host and does a tree traversal to locate the parent directory file.

2) If the parent directory file is not found or found but write access to the parent directory file is not allowed, an appropriate error code is returned ("file not found" or "write not permitted").

3) If the directory file is found, and room exists in the directory, the new file is entered in a branch. As mentioned above, no duplicate file names will be allowed by the Supervisor.

CREATE_LINK <pathname, link ,userid>. This command requests that the Supervisor create a link in the specified directory under the specified file name. As already mentioned, the Supervisor will not allow links to form loops. This is done by restricting the maximum number of files in one pathname to 64 files. (This figure is reached by allowing a maximum pathlength of 128 bytes and having file names of one character. File name delimitors of one character, viz. ">", will give a maximum pathlength of 64 files.) By keeping track of the path traversed, the Supervisor is able to determine if and when a loop is formed. Actions taken:

1) The Supervisor locates the root of the virtual file system for this Host and does a tree traversal to locate the parent directory file.

2) If the parent directory file is not found or found but write access to the parent directory file is not

allowed, an appropriate error code is returned.

3) If the parent directory file is found and room exists in the directory, the link is entered in a link entry.

DELETE_FILE <pathname ,userid>. This command requests that the Supervisor delete the specified file from the virtual file hierarchy. For design simplicity, only terminal files (including msf's), can be deleted. This means that directories must be empty in order to be deleted. Actions taken:

1) The Supervisor locates the root of the virtual file system for this Host and does a tree traversal to locate the parent directory file.

2) If parent directory file is not found or found but write access to the parent directory file is not permitted, an appropriate error code is returned.

3) Otherwise, if the file is located, it is deleted by the Supervisor.

READ_FILE <pathname, command_type(directory, data, size) ,userid>. This command requests that the Supervisor transmit to the Host either a data file, directory file (selected elements only), or the File_Size, Last_Update, and Access_Class (entry data) elements associated with a particular file. An explanation of the last parameter, to transmit entry data only, needs some explaination.

Branch entry elements can be logically divided into

two categories with respect to discretionary security. The first category, which includes Entry_Name, Branch_Link_Switch, Access_Class, and ACL_Ptr are branch entry attributes which cannot be altered by a Host process unless the process has discretionary write access to the directory which contains the file branch entry.

The second category, which contains File_Size and Last_Update, are attributes which 'belong' to the file and must be updated when the file is updated. A situation may exist where a process may not have any discretionary access to a directory but may have discretionary read access to a file in the directory (plus implicit access to the rest of the directory during the "search"). In order to read this file, the Host system will need to know file size in order to prepare to receive it. This is the situation where the READ_FILE (size) command is needed. Actions taken: (for data file)

1) The Supervisor locates the root of the virtual file system for this Host and does a tree traversal to locate the desired directory file.

2) If the file is not found or found but read access to the file is not allowed, an appropriate error message is returned.

3) Otherwise, the file is transmitted to the requesting Host System.

(for directory file)

1) Same.

2) Same.

3) If the directory file is found and read access allowed, selected elements of the branch/link entries are returned to the Host.

(for file size)

1) The Supervisor locates the root of the virtual file system for this Host and does a tree traversal to locate the desired file.

2) If the file is not found or found but read access to the file is not permitted, an appropriate error code is returned.

3) Otherwise, the File_Size and Last_Update elements are returned to the Host.

STORE_FILE <pathname, file_size ,userid>. This command requests that the supervisor store the specified file in the FSS. Actions taken:

1) The Supervisor locates the root of the virtual file system for this Host and does a tree traversal to locate the data file.

2) If the data file is not found or found but write access to the data file not allowed, an appropriate error code is returned. Note that Host systems can store only data files; directories are 'built' by the Supervisor.

3) Otherwise, a store operation is performed by the FSS.

READ_ACL <pathname ,userid>. This command is used by

the Host systems in conjunction with the ADD_ACL_ENTRY and DELETE_ACL_ENTRY to adjust (give/rescind) the access mode (read/write) allowed to a Host/Host user to a specific file. Actions taken:

1) The Supervisor locates the the root of the virtual file system for this Host and does a tree traversal to locate the parent directory file.

2) If the file is not found or is found but read access is not allowed to the parent directory file, an appropriate error code is returned.

3) Otherwise, the supervisor returns the file ACL for Host system user examination.

ADD_ACL_ENTRY <pathname, ACL_Entry ,userid>. This command requests the Supervisor to add to the specified file ACL the specified ACL_Entry (Host.user combination plus associated access mode). As with the previous commands, the access is checked for correctness by both the Supervisor and the Kernel before any action is taken.

DELETE_ACL_ENTRY <pathname, ACL_Entry ,userid>. This command requests that an ACL_Entry be deleted from a file ACL. Again, appropriate discretionary and non-discretionary checks are made before any action is taken by the FSS.

ABORT. This command requests the Supervisor to quit execution of the present command and return the file system to its original state. There are only certain locations in the execution of Host commands that the Supervisor is able

to interupt. If an ABORT command is received after an operation has been completed but before the final Host acknowledgement is sent, the original command completion will be acknowledged and the abort command will be ignored. Otherwise, action of the command will be halted and the Supervisor will wait for another Host command. All Host commands (including ABORT) will be explicitly acknowledged with either a "command complete" message or an appropriate error code.

C. PROCESS STRUCTURE

There are two Supervisor processes which act on behalf of each Host system (hardware port). The input/output (IO) process and the file management (FM) process. The IO process is responsible for communication and data transfer (via packets) between the Supervisor and the Host system. The FM process is responsible for managing the per-Host virtual file systems and providing overall FSS control. All Host commands are interpreted by the FM process; the IO process acts in a "slave" mode to the FM process. Acting together, the FM and IO processes interpret and execute the file management requests of the Host systems. Kernel primitives READ, ADVANCE, AWAIT, and TICKET used in conjunction with eventcounts and sequencer (described later), are used to synchronize Host surrogate process execution.

Both the FM and IO processes call on Kernel primitives to perform actual segment manipulation. The normal order in

which these calls are made is fixed by the Kernel design. To add a segment to a process memory, the order of Kernel calls is: 1) Gatekeeper.Create_Segment, 2) Gatekeeper.Make_Known, and 3) Gatekeeper.Swap_In. To delete a segment from a process memory, the order of Kernel calls is: 1) Gatekeeper.Swap_Out, 2) Gatekeeper.Terminate, and 3) Gatekeeper.Delete_Segment. The Supervisor procedures use these invokation orders.

There are three levels of abstraction for a Host surrogate process. They are: 1) the level at which Host commands are known, 2) the level at which files are known, and 3) the level at which Supervisor segments or packets are known. These levels of abstraction should be kept in mind when reading the FM and IO process descriptions.

A design choice to simplify file system maintenance and control is to allow upgrading of only directories (e.g., unclassified to secret). This will eliminate the possibility of having a secret file in an unclassified directory, a situation which would prevent updating of the file branch data by the secret process since writing "down" is not allowed. This restriction is not felt to exclude any significant FSS capabilities and provides for a simplified implementation.

The modular construction of the FSS enhances System structure. All data bases, except the files themselves, are module local. Code is expected to be written in PLZ/SYS [Snook], which is a high level pascal-like structured

programming language. Because of the its length, code is located in Appendix C. The code listed in this appendix gives the interprocess and intermodule control structure of the FSS.

1. Shared Segment Interactions

Supervisor process execution occurs in a completely asynchronous manner. When a process is refered to in the following discussions, the two Host surrogate processes are being referenced; these surrogate processes have the same clearance levels as the Host they represent.

As already mentioned, the task of the FSS is to provide a service. To be of maximum benefit, this service should be unambiguous, easy to use, and robust.

The major problem that the FSS must handle for proper System security is the confinement problem, viz., to prevent a process from reading a file with a higher classification or writing (i.e., storing or updating) a file with a lower classification. This job is handled entirely by the Kernel.

Another problem closely related to the confirement problem which also irvoles the Supervisor, is the "readers/writers" problem [Courtois]. In order to preserve file integrety, reading and writing of a shared file cannot be allowed at the same time. Since a primary objective of the FSS is to provide for the sharing of files, this problem will certainly occur and must be handled properly for System

viability.

Both the confinement problem and the readers/writers problem can be solved in one of two ways. Mutual exclusion, a mechanism which forces a time ordering on the execution of critical regions, forces concurrent processes into a total order execution sequence. This is counterproductive to the purpose of a process structure, which inherently allows concurrent execution of processes.

A second and relatively new method is the use of eventcounts and sequencer [Reed] to control access to critical regions. This method preserves the idea of concurrent processing to a much greater extent. An eventcount is a object that keeps count of the number of events (in the case of the FSS, segment read/write accesses) that have occured so far in the execution of the System procedures. These eventcounts are associated with the Supervisor segments. They are accessed only via Kernel calls and can be thought of as non-decreasing integer values. Each Supervisor segment has two eventcounts associated with it, one to keep track of the read accesses and one to keep track of the write accesses.

A Kernel primitive ADVANCE signals the occurrence of an event (read/write segment access) associated with a particular segment eventcount. The value of an eventcount is the number of ADVANCE operations that have been performed on it. A process can observe the value of an eventcount by either READ(Seg_#, E), which returns the value directly, or

by AWAIT(Seg_#, E, t), which returns when the eventcount reaches the specific value t.

A sequencer is also necessary to solve the confinement and readers/writers problems. Some synchronization problems require arbitration (e.g., two write accesses to the same segment); eventcounts alone do not have the ability to discriminate between two events that happen in an uncontrolled (i.e., concurrent) manner. A sequencer, like eventcounts, can be thought of as a non-decreasing integer variable that is initially zero. Each Supervisor segment has associated with it one sequencer. The only operation on a sequencer is a Kernel primitive operation called TICKET(Seg_#, S), which, when applied to a sequencer, returns a non-negative integer value. (Similar to getting a ticket and waiting to be served at a barber shop.) Two uses of TICKET(Seg_#,S) will return two different values corresponding to the relative 'time' of call.

The segment number associated with these synchronization primitives informs the Kernel of which segment is being refererced. The use of eventcounts and sequencer can be illustrated by examining the following two procedures (read <> as not equal). The FSS implements these functions in the Directory_Control module located in the FM process.

```
PROCEDURE reader
  BEGIN INTEGER w;
abort:  w := READ(Seg_#,S); !get reader eventcount!
        AWAIT(Seg_#,C,w); !wait until write complete!
        'read file';
        if READ(Seg_#,S) <> w THEN GOTO abort!read again!
  END

PROCEDURE writer
  BEGIN INTEGER t;
        ADVANCE(Seg_#,S); !increment reader eventcount!
        t := TICKET(Seg_#,T); !get sequencer!
        AWAIT(Seg_#,C,t); !wait for write to complete!
        'read and update file';
        ADVANCE(Seg_#,C); !increment writer eventcount!
  END
```

The Kernel will enforce the confinement property and prevent the application of the ADVANCE and TICKET primitives to segments with an access class less than the Host access class. Not to do so, would allow a communication path to be created between two different access levels. The two eventcounts a Supervisor segment will have associated with it (in the Kernel) are a write eventcount, C, and a read eventcount, S. Each segment will also have a sequencer, T, associated with it. Eventcounts and sequencer are initially zero.

These eventcounts and sequencers, with their associated Kernel primitives, are used by the FSS to perform the synchronization functions of Block and Wakeup [Coleman], described in the original Kernel design. Eventcounts and sequencers provide a clearer picture of the process interaction as well as explicit control of the 'readers/writers' problem. Even more importantly, they

permit the synchronization between processes of different access levels. This is essential in order to permit a high level Host to read files of a lower level.

There are two groups of Host requests. They can be classified as read requests (e.g., READ_FILE, READ_ACL) and write requests (e.g., CREATE_FILE, STORE_FILE). These categories can be further subdivided into read data file, read directory file and write data file, write directory file subcategories. Each category type must be handled in a proper manner by the Supervisor to insure file integrity. Each category will be discussed in turn beginning with the read file category.

There two conditions which might develop over which a process has no control; file update by another process, and file deletion by another process. An example of file update might occur while a secret process is traversing a file hierarchy and is in the middle of searching the directory for an Entry_Name when another process (at the directory access level) updates the directory. Since the secret process will READ the segment "reader" eventcount, S, before and after the search, it will know that the data it had obtained is possibly invalid. Although there does not appear to be a problem with allowing the 'reading' process to re-read the directory file until a "good" read is achieved, a closer examination of this condition should be made at implementation time, viz., is it possible for a 'writing' process to alter the pathname of a 'reading'

process so that an inconsistant state is achieved for the reading process? A possible solution could require a process which suffers a "bad" read to begin the traversal over, beginning at the root directory.

When a directory is being read to pass directory data back to a Host, the directory data is put in a buffer and sent from there.

A single segment buffer may be to small to hold a data file (e.g., maximum file size of 256K bytes). Therefore, to present the Host with only valid data, a data file "buffer" is needed at the process level. Since this buffer will be at the process access level, it can be locked by the process to insure that no other process interfers during the reading operation once the data file is in the buffer file. This copying of the data file is done by the FM process and the IO process will read the file from the buffer file when transfering the file to a Host system. The choice of making a copy of a data file is awkward but considered necessary in order to provide the Host with only atomic operations, i.e., to prevent the situation from occuring where half of a ten segment msf is transmitted to the Host and the file is either updated or deleted.

The other condition which may arise during a file read is a file deletion. This situation occurs when one process is reading a file and another process deletes the same file. The first process, not knowing that the file (segment) has been deleted, will try to reference the file

again. A hardware segment fault will occur and cause a transfer of control to the Kernel. Note that in this situation, it is the higher access class process which will suffer the fault while it is reading a lower access class file. To handle this problem, viz., the Supervisor segment fault, a fault handler must be part of the distributed Supervisor. A Kernel primitive also needs defining. This primitive, Gatekeeper.On_Fault (Fault_condition, Entry_pt), is called in the initialization of the Supervisor process where it is possible for a segment fault to occur. A call to a Superivsor condition establisher is also necessary. This will place a specific condition handler on a 'condition stack". If a fault occurs, the Kernel returns to the Supervisor fault handler with a 'segment fault' error condition. This fault handler in turn transfers control to the condition handler at the top of the 'condition stack' which can make a normal return from all procedures. When the error condition is detected (from the return code) by the appropriate Supervisor level, action is taken, viz., the Host command in re-initiated. Since the file (segment(s)) has been deleted, this reinvocation may well result in a 'segment not found' error condition being returned from the Kernel and a "file not found" error condition being relayed to the Host. When the Supervisor exits the "segment fault" a "revert" command is necessary to remove the condition handler from the condition stack.

Another side benefit of having the Supervisor do all

the actual file reading (and therefore take all the segment faults) is that it prevents a hardware fault from occuring during the actual data transfer in the Kernel during IO process execution; this condition would force the handling of the fault in the Kernel domain--a difficult task.

Writing a file is a more straight foreward task and presents fewer problems. This is because a writing process has the same access class as the file and can prevent all other access to the file (segment(s)) it is concerned with. To alter a directory (CREATE_FILE, DELETE_FILE, etc.), a process will get a ticket to the directory and perform the necessary manipulation when its number is called. In order to store a file, more care must be taken. If a process were allowed to store directly into the old file, the possibility exists that a software or hardware error might result in a partially updated file and loss of file integrity. To prevent this from occurring, a data file is first stored into a temporary file set up by the FM process. This also allows the original file to continue to be read by other processes while the store operation is going on, a significant advantage if the data file is long. After the file is stored by the IO process, the FM process gets a ticket to the file directory and when its turn comes, makes the necessary directory updates, viz., the temporary file name is subsituted for the old file Entry_Name, Last_Update information changed, and the old file deleted. (If the file is a msf, each segment is, of course, deleted.)

2. File Management Process

The FM process is composed of the five modules depicted is figure 8 (with associated Kernel calls). The FM process is the controller of the FSS and directs all interaction between the FSS and a Host system. Each module which makes up this process will be described along with the procedures which make up the individual modules.

a. File Management Command Handler Module

As depicted in figure 8, the FM_Command_Handler module (see Appendix C, p. 104) is at the top of the FM process hierarchy. This is the level of abstraction at which Host commands are "known". This module is responsible for interprocess communication and synchronization (with the IO process) and Host command interpretation. Interprocess communication is achieved by the Kernel primitives TICKET, ADVANCE and AWAIT which act on an eventcount associated with the shared mail_box segment. Figure 9 shows the logical construction and the data base description of the mail_box. Figure 10 is a list of the procedures contained within the FM_Command_Handler module and their input and output parameters.

The FM_Cmd_Hnd procedure is the entry procedure into the FM_Command_Handler module. This is the control procedure of the module and is responsible for routing Host commands to specific FM_Command_Handler procedures for action. When notified by the IO process that a command

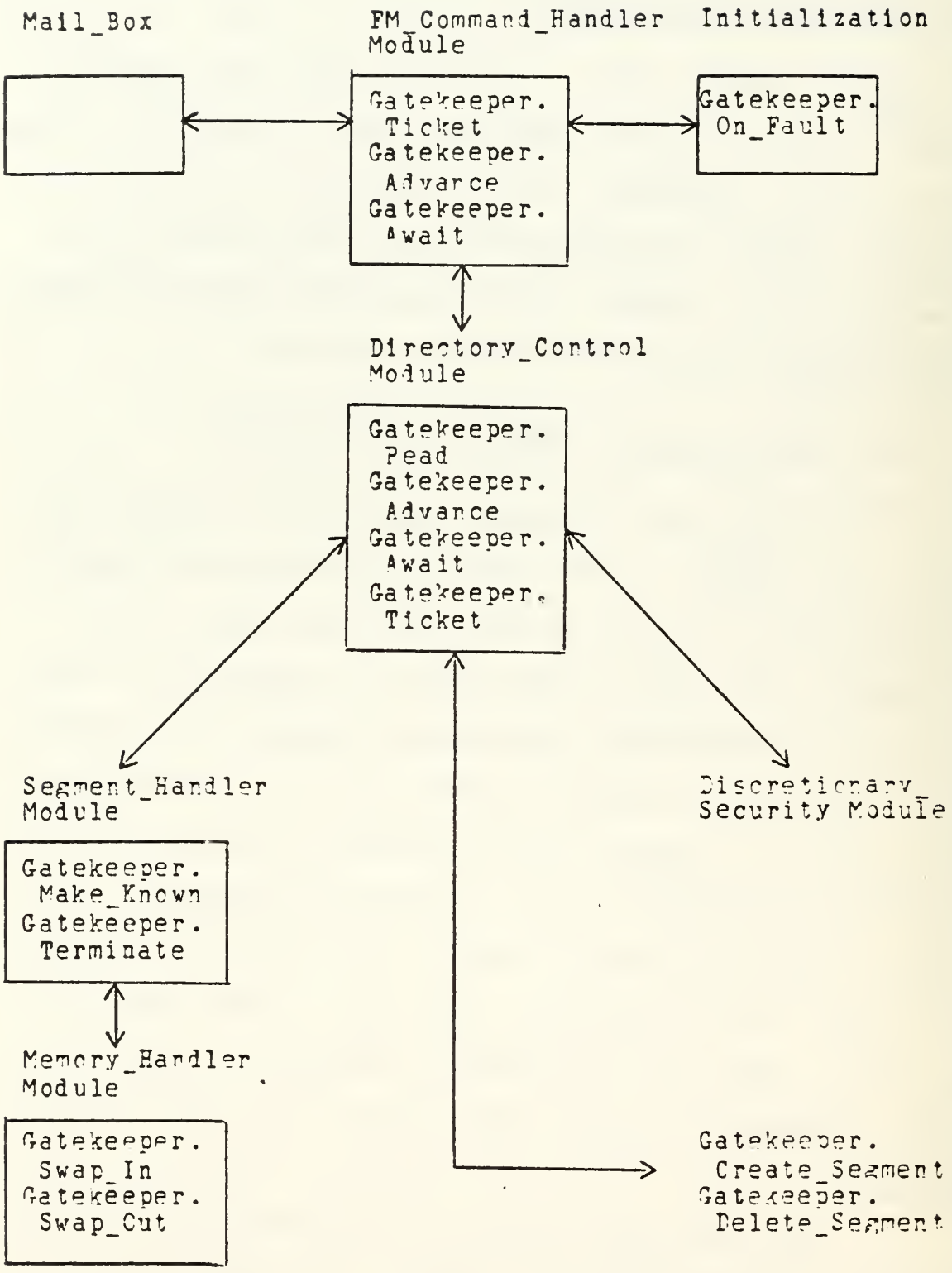
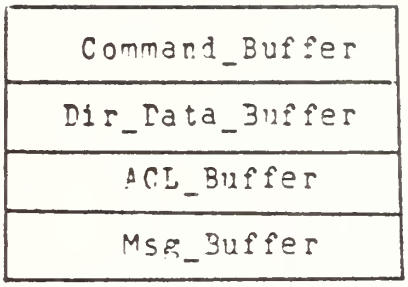


Figure 8. FM Process Modules



Mail_Box Segment

```
Mail_Box Record [

     Command_Buffer      Array   [* bytes]
     Dir_Buffer          Array   [Max_Entry]     Dir_Data
     ACL_Buffer          Array   [Max_ACL_Size]  ACL_Entry
     Msg_Buffer          Record  [Inst           byte
                                  Pathname       string
                                  File_size      lword
                                  Success_code   byte]

                    ]
```

Figure 9. Mail_Box Segment

PROCEDURE	INPUT	OUTPUT
FM_Cmd_Hnd	Host Cmd	Mail_Box.Msg.Inst Mail_Box.Msg.Succ.Code
FM_Cmd_ Delete_File	Pathname Userid	Mail_Box.Msg.Inst Mail_Box.Msg.Succ_Code
FM_Cmd_ Create_File	Pathname File_Type Userid	Mail_Box.Msg.Inst Mail_Box.Msg.Succ_Code
FM_Cmd_ Create_Link	Pathname Link Userid	Mail_Box.Msg.Inst Mail_Box.Msg.Succ_Code
FM_Cmd_ Read_File	Pathname File_Type Userid	Mail_Box.Msg.Inst Mail_Box.Msg.Succ_Code Mail_Box.Msg.File_Size
FM_Cmd_ Store_File	Pathname File_Size Userid	Mail_Box.Msg.Inst Mail_Box.Msg.Succ_Code Mail_Box.Msg.File_Size
FM_Cmd_ Read_ACL	Pathname Userid	Mail_Box.Msg.Inst Mail_Box.Msg.Succ_Code Mail_Box.Msg.File_Size
FM_Cmd_ Add_ACL_ Entry	Pathname ACL_Entry Userid	Mail_Box.Msg.Inst Mail_Box.Msg.Succ_Code
FM_Cmd_ Delete_ACL_ Entry	Pathname ACL_Entry Userid	Mail_Box.Msg.Inst Mail_Box.Msg.Succ_Code

Figure 10. Command_Handler Module
Procedure Input/Output parameters

packet is in the mail_box, the FM process retrieves the command and begins appropriate action. The Host command (e.g., STORE_FILE, READ_FILE) is actually an entry into a case statement which directs the correct FM_Command_Handler procedure to take action. Each Host command has associated with it, at this level, its own procedure.

Because the procedures of the module are relatively straight forward, they will not be discussed in detail. The general functions of all the procedures in this module are to pass instructions to the IO process and to the Directory_Control module, the "workhorse" of the FM process.

Some explanation of Host command parameters is in order, however. These parameters (described below) are:

pathname

link

file type

command type

file size

access level

userid

ACL entry.

In all host commands, the pathname passed by the Host is the pathname (relative to the 'root' directory of the Host virtual file system) of the file of interest, whether a directory or data file. From the pathname, the FM process is able to extract the pathname of the parent directory which it must bring into the FM process memory to

check for proper discretionary access. The complete pathname, in terms of the FSS file system, is passed to the Directory_Control module for actual directory manipulation. A pathname and file size (for the 'buffer file') is returned (dir_pathname, dir_file_size) by the Directory_Control module during a Host READ_FILE or STORE_FILE request. This new pathname and file size is passed to the IO process where the actual data transfer takes place for these operations. Since discretionary security checks are made in the FM process and all input/output "buffers" (e.g., temporary data file, mail_box segment) are under positive FM process control, the IO process need not be concerned with discretionary security or the possibility of a "segment fault".

A link is a pathname which a Host passes in the CREATE_LINK command.

File type is used for the CREATE_FILE Host command and is necessary because of the different file formats.

Command type is used in the READ_FILE Host command to specify the type of 'read' the FSS is to conduct, i.e., to read a directory file, a data file, or only a data file size.

File size is passed by the Host during STORE_FILE requests. This information is necessary for the FM process to create a temporary file of sufficient size to store a Host file. File size is relayed to the IO process so

that the IO process can go directly to the data file without having to check the directory file for file size. File size is in bits.

Access level is needed for the CREATE_FILE command. This allows for upgraded directories (remember, data files cannot be upgraded).

The identification of the Host system user is necessary for the FSS to perform discretionary security checks. This is provided by the Host system through the userid parameter.

ACL_Entry is used with the ADD_ACL_ENTRY and DELETE_ACL_ENTRY commands to give/rescind discretionary access to files.

b. Directory Control Module

The Directory_Control module, as the name implies, does the directory manipulation and maintenance. Figure 11 lists the procedures which make up this module along with their input/output parameters.

This is the level of the FM process at which files are known. The Directory_Contorl module handles the readers/writers problem with the appropriate use of the Kernel syncronization primitives READ, ADVANCE, AWAIT, and TICKET. It handles the segment fault condition by a call to the condition establisher when the possibility of a segment fault exists. The IO process uses the same primitives while performing its portion of the data file read and store

PROCEDURE	INPUT	OUTPUT
Dir_Cntrl_ Directory	Command_Type Userid Pathname File_Type Access_Level Link ACL_Entry	Dir_Succ_Code
Dir_Cntrl_ Data	Command_Type Userid Pathname File_Size	Dir_Succ_Code Dir_Pathname Dir_File_Size
Dir_Cntrl Update	Command_Type Userid Pathname	Dir_Succ_Code

Figure 11. Directory_Control Module Procedures Input/Output Parameters

operations, viz., the tree traversal when locating the data file read buffer or the temporary storage file. As previously mentioned, the IO process will not face the problem of file deletion while reading and will therefore not have to establish a condition handler.

Logically, Host requests require four basic actions to be performed at this level. They are: 1) to bring a directory file into process memory for a read and/or write operation, 2) to delete a file, 3) to create a file, or 4) to copy a data file into a data file buffer. All other file mainterance functions such as managing memory or managing the limited number of segments available to a process, are performed by subordinated modules. There are three procedures in this module.

The Dir_Cntrl_Directory procedure is the Directory_Control module procedure which handles Host commands which require that the parent directory be brought into process memory in order that required discretionary security checks can be made. These Host commands are:

DELETE_FILE

CREATE_FILE

CREATE_LINK

READ_FILE (dir, size)

READ_ACL

ADD_ACL_ENTRY

DELETE_ACL_ENTRY.

To perform these tasks, the parent directory

segment (which contains the file branch/link entry) must be brought into process memory to check for proper discretionary access. If access is permitted, the Segment_Handler module is called with a pathname of a segment required to be brought into process memory.

For action on a DELETE_FILE command, discretionary write access to the directory is required since the branch/link entry of the file must be removed from the directory when the file is deleted. (Note that this raises the possibility of a Host having write access to a file but not able to delete it because he does not have write access to the directory.) If the parent directory file is not found or found but write access to the directory not permitted an appropriate error code is returned, viz., "file not found" or "write access not permitted".

If an error condition does not arise, the directory is brought into process memory and a check of the file attributes is made to determine file type (data, directory, link). If it is a data file or link entry, it can be deleted because it is a terminal node in the file hierarchy. If it is a directory, the (directory) file itself must be brought into process memory to see if the directory is empty (viz., check of Entry_Count and presence of a Supervisor temporary file). If it is not empty, an error code of "not terminal file" is returned to the Host. If the directory is empty, it can be deleted.

If no error condition occurs during the

preceding checks, the file may (subject to check by the Kernel) be deleted. The Dir_Cntrl_Directory procedure will call on Seg_Fnd_Make_Unaddressable procedure which will in turn call Mem_Hnd_Swapout procedure to remove the segment from process memory if it is in memory. (Remember the actual order: Swap_Out, Terminate, Delete.) Next, the Kernel primitive, GateKeeper.Delete_Segment is called to delete the file from the FSS. Note that in the case of msf's, these steps must be repeated until all segments of the file are deleted. At this time, the branch entry is removed from the directory by zeroing all branch entry elements (to allow for Kernel secondary storage compaction of disc pages of zeros). The IO process is then instructed to acknowledge the Host with "file deleted". This frees the entry for future use.

The deletion of a link requires the same discretionary write access to the directory. In this case, no further checks are necessary and the link entry elements are zeroed in the directory, freeing the entry for re-use.

For the CREATE_FILE command, analogous action is taken by the Dir_Cntrl_Directory procedure, viz., to check discretionary write access to the directory which will contain the file branch entry.

Once this check has been satisfactorily completed, and room exists in the directory, the Kernel call Gatekeeper.Create_Segment is made to create the file. The initial file size is zero for data files since the Supervisor has no prior knowledge of the size of the file

that will be stored in the branch entry. As explained earlier, a file size of LARGE (8K bytes) was selected for the fixed directory size.

The CREATE_LINK request is again analogous, the only difference being that instead of a branch entry being made in the directory, a link entry is made. As previously mentioned, the Supervisor will not allow a loop state. Checks will not be made at link creation time; however, the Supervisor will 'abort' a file search if it encounters this error condition during tree traversal.

The READ_FILE (dir) command requires read access to a directory file. If no error condition arises during discretionary security checks, selected directory data (e.g., Entry_Name, File_Size, etc.) is transfered to the Host system via the mail_box segment (viz., Dir_Data_Buffer). This selected directory data for each 'occupied' branch/link entry is transfered during the READ_FILE (dir) command. For the READ_FILE (size) request, only selected directory data for a specific data file is transfered. The IO and FM processes use appropriate Kernel synchronization primitives to assure that the information in the mail_box segment is valid.

The last three Host requests handled by the Dir_Cntrl_Directory procedure are related. Again, appropriate discretionary access checks must be made in the parent directory. If no error condition arises, the action taken is straight foreward. In the case of the READ_ACL

command, the file ACL is transfered to the mail_box ACL_buffer and the procdure returns to the FM_Command_Handler module. In the case of the ADD(DELETE)_ACL_ENTRY commands, the action is completed by the Dir_Cntrl_Directory procedure and the appropriate Dir_Succ_Code returned.

The Dir_Cntrl_Data procedure is responsible for transfering to/from a Host a requested data file if necessary preconditions are met (viz., discretionary and non-discretionary security). In order to read or store a file, a Host must have the proper discretionary access to the file. To check this, the parent directory which contains the file branch entry must be brought into memory. This is done by the Segment_Handler module. If the proper access is not allowed, an error code is returned to the FM_Command_Handler module for relay to the Host system. If the proper access is allowed, a copy of the file is made in the case of the READ_FILE command, or a temporary file is created in the case of the STORE_FILE command. The pathname and file size of the data files to be transfered are passed to the IO process which will perform the actual data transfer. Upon a successful transmission of the data by the IO process, the FM process instructs the IO process to acknowledge the Host with a "read complete" or "store complete", as appropriate.

The Dir_Cntrl_Data procedure will make appropriate use of Kernel synchronization primitives (e.g.,

AWAIT, READ, etc.) when copying a data file into the data file read buffer or setting up a temporary file for the store operation. After the file transfer has taken place in the IO process, the IO process returns a success code to the FM process. The IO process will return to the FM process when one of three conditions exist: 1) either the read or store operation is successful and complete, or 2) a command packet is received (viz., an abort command), or 3) a 'time-out' occurs and the IO process was not able to complete the command.

For a store operation, the Dir_Cntrl_Update procedure is called to update the directory data (viz., exchange the temporary file Entry_Name with the old file Entry_Name) and deletes the old file. (The temporary file should be deleted by this procedure if, upon attempting to update the file, the old file cannot be found.)

Since each directory segment has only one temporary file for file update, some delay may be experienced by Host systems if several try to store large files into the same directory. This does not appear to be a major problem since most users are anticipated to be operating from their own directory files.

The Dir_Cntrl_Update procedure is also used to free the temporary storage file in the case of a Host abort command.

c. Discretionary Security Module

The Discretionary_Security module is responsible for checking Host user discretionary access to a specific file and adding and deleting ACL_entries. All file ACL's are logically located in this module. This is the only other module besides the Directory_Control module where a segment fault might occur. Appropriate use of the condition establisher must be made before any attempt to read an ACL so that a proper return is executed to the Directory_Control module in the event of a fault. There are four procedures which make up this module as depicted in figure 12.

The Disc_Sec_Check_Access procedure, as the name implies, checks for a specific user discretionary access to a specific file. A success code returns, indicating the result of the check. This discretionary check is only made on the specific file which is required in a Host command, i.e., a design choice was made not to make discretionary access checks during the tree traversal search for the specified file. This makes explicit in one ACL who has access to a file, which contributes to clear security semantics. (This also eliminated the question of what to do if an intermediate directory was encountered during a file search to which the process did not have read access.)

The Disc_Sec_Add_ACL_Entry procedure adds an ACL_entry to a file ACL and returns a success code to indicate the action taken. As noted previously, a directory has a limited number of ACL_entry elements. The Supervisor only guarantees one ACL_entry element per branch entry (for

PROCEDURE	INPUT	OUTPUT
Disc_Sec_ Check_Access	ACL ACL_Entry Userid	Disc_Succ_code
Disc_Sec_ Add_ACL_Entry	ACL ACL_Entry Userid	Disc_Succ_Code
Disc_Sec_ Delete_ACL_Entry	ACL ACL_Entry Userid	Disc_Succ_Code
Disc_Sec Get_Entry	ACL_Entry Userid	Disc_Succ_Code

Figure 12. Discretionary_Security Module
Procedure Input/Output Parameters

the file creator). If another ACL_entry is required and the ACL_entry "pool" is empty, an ACL_entry element will have to be explicitly freed from a file by the Host before a file ACL can be added to.

The Disc_Sec_Delete_ACL_Entry procedure performs the straight foreward task of deleting an ACL_entry from a file ACL. This procdure returns a success code when deletion is complete.

The last procedure of this module is the Disc_Sec_Get_ACL procedure. It is used during the intial creation of a file by the Directory_Control module to get an initial ACL_Entry element.

d. Segment Handler Module

The Segment_Handler module is the abstraction level at which Supervisor segments are known. This module works in conjunction with the Memory_Handler module (described later) to either bring a segment into process memory (viz., Make_Known, Swap_In) or to terminate a segment (viz., Swap_Out, Terminate). This module is responsible for maintaining the PM_KST (known segment table--figure 13) data base. The data base elements of the PM_KST are the pathname of a segment known to the process, the segment number (Seg_#) of the terminal file in this pathname, mode (i.e., read or write), and the use bit necessary for a LRU removal algorithm (approximation). To prevent the situation where a segment has been deleted by one process but is still

Pathname	Seg_#	Mode	Use

Figure 13. FM_KST

PROCEDURE	INPUT	OUTPUT
Seg_Fnd_ Make_Addressable	Pathname	Seg_# Seg_Succ_Code
Seg_Fnd_ Make_Unaddressable	Pathname	Seg_Succ_Code

Figure 14. Segment_Fandler Module Procedure Input/Output Parameters

indicated as "in memory" by another process, each new Host command will initiate a Kernel call, Gatekeeper.Swap_In (Seg_#, Base_Addr), to confirm the existence of a segment. A Kernel return of "segment not found" will indicate that the segment has been deleted. The FSS must then clear its data structures of invalid data and traverse the virtual hierarchy from the root directory to insure that the segment is truely gone and that it has not been renamed by another process i.e., to cover the unlikely situation where a pathname has been deleted and then re-created with the same filenames. This would asscciate different segment numbers with the same pathname.

Figure 14 is a list of the procdures of this module along with their input/output parameters. This module receives a file segment pathname and returns when it has been brought into process memory or an error condition arises. The possible error condition that might be returned from this module is 'file not found'. This module has two tasks, and therefore two procedures. To make a segment addressable by the Host process (viz., bring it into process memory) or to make a segment unaddressable by a Host process (viz., to remove the segment from process memory). The procedures which handle these tasks are the Seg_Hnd_Make_Addressable (i.e., bring a segment into process memory) and Seg_Hnd_Make_Unaddressable (i.e., remove a segment from process memory) procedures. (Note that to make a segment addressable also requires making the segment

"known" and that making a segment unaddressable requires "terminating" the segment.) Both tasks are accomplished by appropriate use of Kernel primitives and accompanied by calls to the Memory_Handler module to Swap_In or Swap_Out a segment.

This module is also responsible for segment management. Segment management is necessary because each MMU allows the addressing of only 64 sements. With one MMU in the initial FSS implementation and several segments taken by the Supervisor and Kernel segments, the number available to the Supervisor processes will be somewhat less (MAX_KNOWN_SEG) than 64. This number must be managed in a dynamic manner without interfering with process execution.

The Seg_End_Make_Addressable procedure is the more involved of the two module procedures. If a request to make a segment known is received, the FM_KST is checked to see if it is already known. If it is, the LRU bit is set and the Memory_Hardler module is called to assure that the segment is in process memory. If it is not already known to the process, it must be made known by the Kernel call, Gatekeeper.Make_Known (Par_seg_#, entry_#, mode). But this can only be done if process segment limit is not exceeded. If the addition of a segment will cause an overflow, a segment must be removed by the Seg_End_Make_Unaidressable procedure. Once this is done, the desired segment can be made known, the FM_KST updated, and the Memory_Handler module called to bring it into process memory.

The Seg_Hnd_Make_Unaddressable procedure is straight foreward. This procedure may be called to either delete a specific segment or to delete the LRU segment. If called to remove a specific segment, action is taken to remove the segment (described below). If called to remove the LRU segment, a LPU removal algorithm (approximation) is used to determine which segment will be removed. When this has been done, the Memory_Handler module is called to Swap_Out the segment from process memory. A returned success code indicates that the segment has been removed by the Kernel call Gatekeeper.Swap_Out (Seg_#). A call is then made to terminate the selected segment. The Kernel call, Gatekeeper.Terminate (Par_Seg_#, Entry_K#), will cause the segment to be deleted from the Kernel KST. Removing the segment pathname from the FM_KST will complete the action taken by this procedure.

e. Memory Handler Module

This module operates in a "slave" mode to the Segment_Handler module and consist of two procedures. These procedures are listed in figure 15 along with their input/output parameters. The job of this module is to dynamically manage a fixed size linear virtual memory. It does this by swapping in and out of process memory segments as required.

When the Mem_Hnd_Swap_In procedure is called, the FM_AST, figure 16, (active segment table) is checked to

PROCEDURE	INPUT	OUTPUT
Mem_Hnd_Swap_In	Seg_# Seg_Size	Mem_Succ_Code
Mem_Hnd_Swap_Out	Seg_#	Mem_Succ_Code

Figure 15. Memory_Handler Module
Procedure Input/Output Parameters

Seg_#	Size	Base_Addr	Use

Figure 16. FM_AST

0	1	2	. . .	Base_Addr
				SEG_#

Figure 17. Mem_Map

see if it is already in memory. If it is, its LRU bit is set and Gatekeeper.Swap_In (Seg_#, Base_Addr) is called to insure that the segment has not been deleted by another process since last use. If the segment is not in memory, the MEM_MAP data structure, figure 17, is checked to find room for a segment of the required size. Arguements can be made for both a first-fit and best-fit memory management scheme [Shaw]. A first-fit scheme is chosen for the FSS due to the simpler implementation and the reduced memory fragmentation. If room cannot be found, Mem_Mgt_Swap_Out is called iteratively until enough room exist for the segment to be brought into process memory. A Kernel call, Gatekeeper.Swapin (Seg_#, Base_Addr), is used to move the segment into process memory when room exists.

Mem_Mgt_Swap_Out may either be called to remove a specific segment or to remove the LRU segment from process memory. If the request is to remove a specific segment, the task is straight foreward; a call is made to the Kernel primitive Gatekeeper.Swap_Out (Seg_#). If the request is to remove a specific segment, a LRU algorithm (approximation) is used to determine which segment to remove. When this is done the Kernel call is made and the Memory_Handler data bases are updated to reflect the segment removal.

A preliminary analysis of memory requirements indicates that process linear virtual memory will need to be at least 24K bytes. The driving factor in this calculation is the fact that two data segments (possibly 8K bytes each)

may be required in process memory during the copying of a data file into the data file "buffer". A 24K byte memory would allow for the worst case, viz., one 8K byte segment positioned in the middle of linear memory; room would still exist for the two 8K byte segments.

3. Input/Output Process

The IO process is the second of the two processes which act on behalf of a Host system to provide a requested file management service. The IO process acts in a slave mode to the FM process; it receives its commands from the FM process via the shared mail_box segment described in connection with the FM process.

The IO process is responsible, as the name implies, for all input and output between the Supervisor and the Host systems. The IO process is composed of five modules as depicted in figure 18 (along with Kernel calls). Two of these modules, Segment_Handler and Memory_Handler, are the same modules as described in the FM process and will not be discussed further. Their task is to bring into the virtual memory of the IO process the data segments into and from which Host files are stored or read. Note that since discretionary security checks are done in the FM process, the IO process does not have to repeat these checks.

Direct invocation of the Packet_Handler module from the IO_Command_Handler module is possible to send Host "acknowledgements". If a file is to be read or stored, the

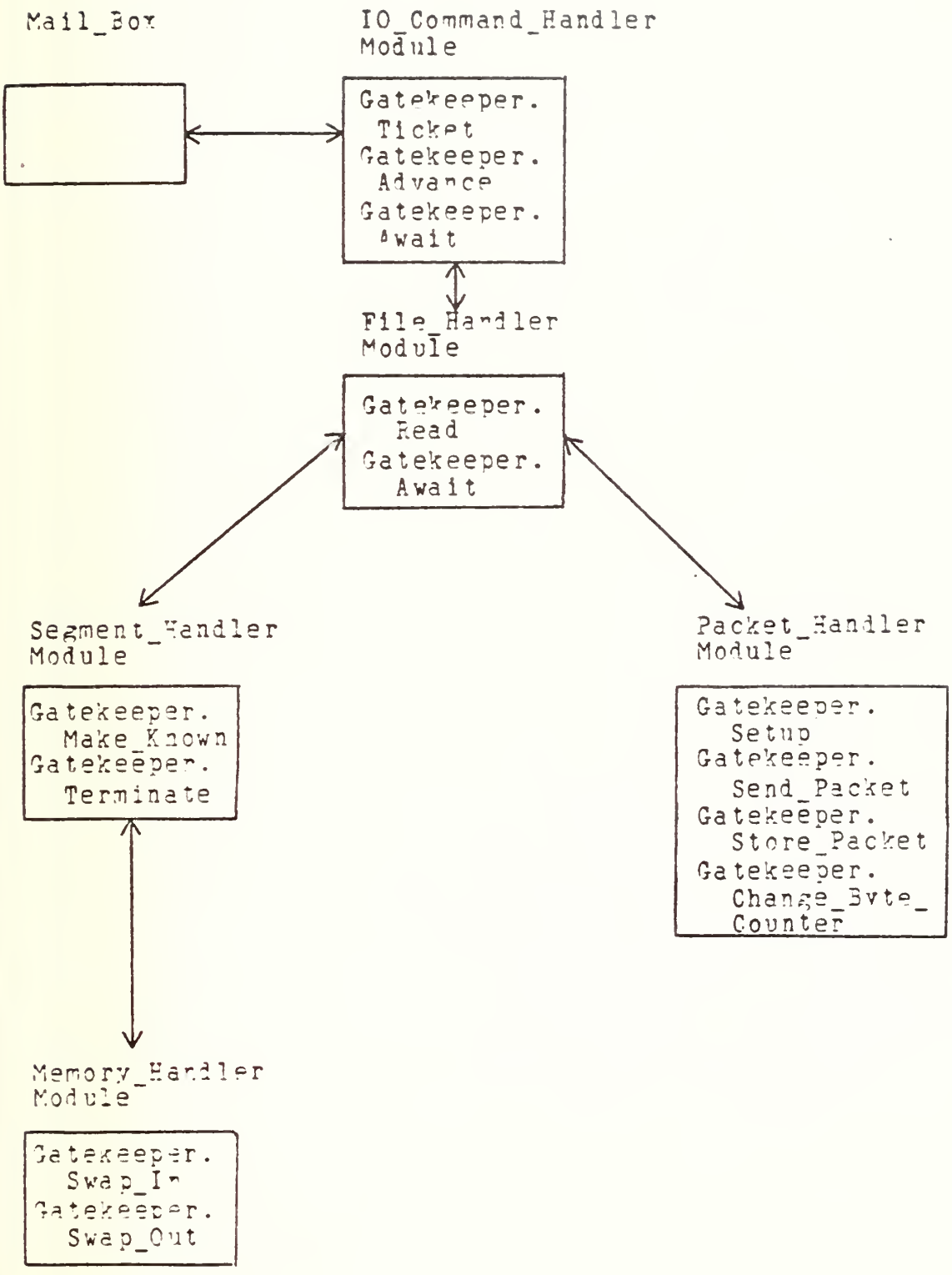
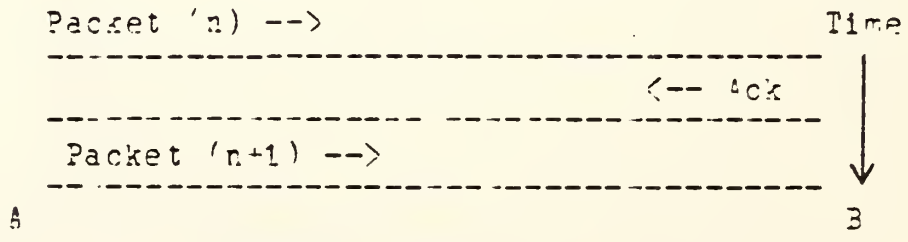


Figure 18. IO Process Module

File_Handler module is first called to perform the read or store operation.

The IO process is also responsible for FSS-Host protocol. Data is transfered between Host and FSS via fixed size "packets". There are three formats for these packets: 1) a synchronization packet format, 2) a command packet format and, 3) a data packet format. Figure 19 gives the logical construction of the data and command packets. The synchronization packet is left for later design in connection with the design for a Host interface. The packet size of 521 bytes for data and command packets was chosen to maximize data transfer efficiency at the expense of increasing the command packet size. Because 512 bytes is the size of the smallest Supervisor segment, this was chosen as the "unit" of data transfer.

A protocol must exist that insures reliable transmission and reception of packets by both the sender and receiver in the FSS-Host packet exchange. The simplest protocol that will handle packet transmission is to transmit packets one at a time and wait for packet acknowledgement before sending the next packet. The following diagram illustrates this simple protocol.



DATA PACKET

Packet_Type	Byte
Packet_Number	Lword
Data	512 Bytes
Check_Sum	Lword

COMMAND PACKET

Packet_Type	Byte
Packet_Number	Lword
Host_Cmd	Byte
Pathname	128 Byte
File_Name	18 Byte
Link	128 Byte
Access_Level	Byte
File_Type	Byte
ACL_Entry	3 Byte
Userid	Byte
Check_Sum	Lword
Padding	231 Byte

Figure 19. Packet Construction

Operating in this fashion is extremely inefficient, especially in the transmission of large data files; it does not allow the sender to send packets before an acknowledgement is received nor does it allow the receiver to accept more that one packet at a time (i.e., read ahead and write behind). A multi-packet protocol is necessary to take advantage of a read ahead and write behind scheme.

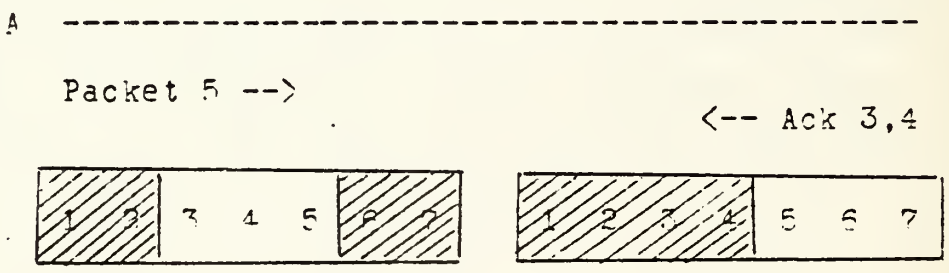
In specifing a multi-packet protocol, some means of distinguishing individual packets must be established. This is done by giving each packet a sequence number carried in the packet header. The receiver returns acknowledgements indicating the sequence number of the packet(s) received and accepted (i.e., no errors detected). The number of packets that may be transmitted before an acknowledgement is received is called the packet "window width". Packet transmission is controlled by an algorithm which uses packet sequence numbers and the window width. At System initialization time and anytime a command packet is received, the sequence number of the FSS is reset to zero. Thus the first sequence number expected by the FSS upon system initiation (and afterwards upon command completion) is zero.

For an explanation of how the packet window works, let N(t) denote the transmitted sequence number of the current packet and let N(t-1) denote the next expected sequence number. The window width is denoted by W. At the start of communication, e.g., when a Host sends a command to

the FSS, the Host is allowed to transmit packets bearing sequence numbers in the range $0 \leq N(t) < W$. The receiver expects the packets to arrive in correct sequential order. As they arrive, packets are checked for correctness (at both the hardware (USART) and software level); an incorrect packet is discarded and may be considered 'lost'. Let the sequence number of a particular correctly received packet be S. If $S=N(t+1)$ (i.e., the expected packet), then the packet is received in the correct sequence and it should be accepted by the receiver and an acknowledgement sent with the proper sequence number (in this case, S) to the sender. If $S<N(t+1)$, then the packet is a repetition of a packet previously received by the receiver; the second transmission may be due to either a lost or delayed acknowledgement. The receiver should generate another acknowledgement and send it to the sender and otherwise ignore the packet. If $S>N(t+1)$, then the packet is ahead of sequence, indicating that an earily packet has been lost; such a packet should be ignored and an "error" acknowledgement sent so the packet can be retransmitted.

The arrival of acknowledgements at the sender also needs to be discussed. As each acknowledgement arrives, the sender can delete the copy it has retained of the corresponding packet. As packets are acknowledged, fresh packets can be transmitted, i.e., when packet 0 has been acknowledged, packet W can be sent. Acknowledgements can get lost in transmission as well as packets. If a received

acknowledgement does not refer to the earliest transmitted packet awaiting acknowledgement, then, in this protocol, the sender may safely delete all packets up to and including that referenced by the acknowledgement. Against each copy of a transmitted packet will be noted a time (i.e., the time-out) by which time the packet must be acknowledged. Failing such an acknowledgement, the packet must be retransmitted with its original sequence number. A packet will only be received in sequential order, so it will be necessary to retransmit not only the earliest unacknowledged packet, but also all later packets. The following figure illustrates this protocol. The queues should be considered as circular with automatic wrap-around.



In this figure, the sender is node A and the receiver is node B. Node A has sent out packets 3,4, and 5, the last of which is still in transit to B. Node B has received all packets up to and including 4. It has just acknowledged 3 and 4 and is ready to accept 5,6, and 7 when they arrive in order. When node A receives acknowledgement for 3 and 4, it will be able to transmit successfully packets 6 and 7.

This protocol insures that packets are handled in

sequential order which will insure that the data is received and stored correctly. It also assures positive control over the receipt and transmission of packets; a necessary requirement to prevent buffer overflow and loss of data.

The Kernel controls all the hardware assets, as explained in Chapter 1. Kernel calls are therefore necessary to transfer packets between the FSS and the Host systems. The format of these Kernel calls are:

Gatekeeper.Setup (Buff_Addr, Mode, Status)

Gatekeeper.Send_Packet (Offset, Status)

Gatekeeper.Store_Packet (Offset, Status)

Gatekeeper.Change_Byte_Counter (#_of_Bytes, Status)

Each hardware port is virtualized into an input and an output port. Each virtual port has associated with it a unit control block (UCB) at the Kernel level. The elements of these UCB's are:

Byte_Counter: This element is used to keep track of the number of bytes that have been transmitted or received. This counter is modulo "packet size" so that once packets are synchronized, they should remain so. It can be altered by the Change_Byte_Counter call in order to get the FSS and Host back into packet synchronization.

Buffer_Address: This is the starting address in the input/out buffer where packets will be placed (incomming) or taken from (outgoing). It is initialized by the Setup Kernel call.

Buffer_Length: This element is the length (in packets) of the input/output buffer. This allows the Kernel to perform automatic wrap around at the end of the buffer.

Window_Width: This element is used by the input port UCB to prevent buffer over flow. Each invocation of Store_Packet will advance the window and allow another packet to be stored into the IO buffer. If a Host system violates protocol by sending too many packets, the Kernel will dump them to a "bit bucket". This element is used by the output port to control the number of packets that the FSS is able to send to a Host before receiving an acknowledgement. Although this parameter (viz., window width) may be different for the various Host systems, it should not change often and can therefore be set at system initialization.

For a store operation (FSS to receive packets), a Setup call is used to set the input UCB base address to the initial storage location in the IO buffer. A Setup call is also required to set the output UCB with the base address in the IO buffer from which acknowledgments will be sent. It should be noted here that the IO buffer in the IO process is the location that packets are checked for errors and "enpacketed" or "depacketed". It is just a intermdiate stop for data and neither the final destination nor origin of data.

Subsequent Kernel calls to Store_Packet will return the location of the next packet in the IO buffer to be

processed. The Kernel will store ahead into the IO buffer during the store operation but will not over write the buffer. That is, each call to the Kernel will indicate that a new packet location is open. The IO process will control which packets (and how many) are sent to the FSS by proper use of acknowledgements (for both correct and incorrect packets).

Two Setup calls are also necessary for a send operation. They again set the virtual input/output ports for the transfer of packets from the FSS to a Host. Subsequent calls to Send_Packet indicate that a Packet is ready to be transmitted. The IO process knows when it can discard a packet by the acknowledgments it receives from the Host system.

The Change_Byte_Counter primitive is used by the synchronization procedure to shift a UCB byte counter in order to bring packet transmission back into synchronization. (Synchronization may be required during a temporary communication interruption or system start up.)

The following is a description of the three "new" modules which make up the IO process.

a. Input/Output Command Handler Module

At the top of the IO process module hierarchy is the IO_Command_Handler module (see Appendix C, p. 117). This module is responsible for the interface with the FM process. Communication between the processes is via the mail_box

shared segment and synchronization is through the use of an eventcount and the Kernel primitives TICKET, ADVANCE and AWAIT. The procedures of this module along with their input/output parameters are listed in figure 20.

The IO_Cmd_Hnd procedure, like the FM_Cmd_Hnd procedure a case statement, routes FM process instructions to a specific IO_Command_Handler procedure for action.

The procedure involved when the Host command is not a READ_FILE or STORE_FILE request is the IO_Cmd_Hnd_Ack procedure. This procedure is able to invoke the Packet_Handler module directly for performing directed (by the FM process) Host acknowledgement and/or data transfer from the shared mail_box segment.

The IO_Cmd_Hnd_Send and IO_Cmd_Hnd_Store procedures are relatively straight forward. They provide the IO-FM process interface required for a READ_FILE or STORE_FILE Host request. Both procedures call the File_Handler module to perform the actual file manipulation.

b. File Handler Module

The File_Handler module is required for file manipulation in the IO process and is the level in the IO process at which files are known. The procedures which make up this module along with their input/output parameters are listed in figure 21. As mentioned above, there are only two Host requests that require the IO process to bring data files into process memory. These are READ_FILE and

PROCEDURE	INPUT	OUTPUT
IO_Cmd_Hnd	Mail_Box.Msg.Inst	Output returned
IO_Cmd_Hnd_Ack	Mail_Box.Msg.Succ_Code	from subordinate modules.
IO_Cmd_Hnd_Send	Mail_Box.Msg.Pathname Mail_Box.Msg.File_Size	
IO_Cmd_Hnd_Store	Mail_Box.Msg.Pathname Mail_Box.Msg.File_Size	

Figure 20. IO_Command_Handler Module
Procedure Input/Output parameters

PROCEDURE	INPUT	OUTPUT
File_Hnd_Send_File	Mail_Box.Msg.Pathname Mail_Box.Msg.File_Size	File_Succ_Code
File_Hnd_Store_File	Mail_Box.Msg.Pathname Mail_Box.Msg.File_Size	File_Succ_Code

Figure 21. File_Handler Module
Procedure Input/Output Parameters

STORE_FILE. Note that since file size is passed from the FM process, and the the access to the data files involved is controlled in the FM process, data file segments can be brought directly into IO process memory and any requirement for the IO process to access directory files (other than tree traversal) is eliminated. Because the terminal nodes in the tree traversal are controlled by the FM process, the paths to these terminal nodes will not be alterable until control is released by the FM process.

The File_Handler module consist of two procedures, File_Hnd_Send_File (for Host command READ_FILE) and File_Hnd_Store_File (for Host command STORE_FILE). Both procedures operate in a similar manner. Upon receiving a pathname and file size from the FM process, these procedures use the Segment_Handler procedures to bring the necessary data file (segment(s)) into process memory. A call is then made to the Packet_Handler module to transfer data from/to specified segments.

The order of events in the reading and storing of data files follows the following sequences. For a READ_FILE operation, the order of actions taken by the Supervisor are:

1) Discretionary and non-discretionary checks are made in the FM process.

2) A copy is made of the data file into a per-process data file buffer.

3) The pathname of the data file to be read

(remember, directory data is read by the FM process) is passed to the IO process along with the file size. The IO process can determine the file size from the file directory but by passing file size to the IO process, this step is eliminated for the IO process.

4) The read takes place in the IO process.

5) The IO process returns to the FM process with a success code of "read complete" or an appropriate error code. The only reason for a read operation to fail in the IO process is the receipt of an abort command from the Host or a 'time out' which would occur if the Host stopped sending for some unexplained reason.

6) The FM process instructs the IO process to acknowledge the "read complete" or to send the appropriate error code. The data file read buffer is then free for further use.

If the operation is a STORE_FILE operation the following steps are taken by the Supervisor:

1) Discretionary and non-discretionary security checks are made by the FM process.

2) A temporary file is created by the Supervisor large enough to store the file in. Appropriate use of the synchronization primitives prevents this temporary file from being used by more than one process at a time.

3) The pathname of the temporary file is sent to the IO process and the IO process stores the file into the temporary file.

4) The IO process returns a success code to the FM process and the FM process updates the directory to reflect the new file (viz., Entry_Name of temporary file is changed to the old file Entry_Name). The old file is then deleted by the FM process.

5) The FM process then instructs the IO process to acknowledge the "store complete". There is no reason a store operation should fail other than an explicit abort request by the Host system or hardware failure.

c. Packet Handler Module

The Packet_Handler module does the actual transfer of data between the FSS and the Host system and is the IO process level at which the concept of "packet" is known. The procedures of this module along with their input/output parameters are listed in figure 22. The tasks that this module must perform are: 1) synchronization of packets, 2) error detection, 3) packet acknowledgement, and 4) transfer of data to/from Supervisor segments. Figure 23 is a finite state diagram of packet transfer.

The synchronization task is performed on the system IPL and whenever packet synchronization is lost thereafter. Error detection and request for retransmission upon error detection are complimentory functions which are performed on every packet received from a Host.

Packet transfer during synchronization procedures is in groups of three. This allows the

PROCEDURE	INPUT	OUTPUT
Pk_Hnd_Sync	Sync Cmd	Packet Sync
Pk_Hnd_Ack	Packet Mail_Box.Msg.Succ_Code	Pk_Succ_Code
Pk_Hnd_Send	Data	Packet
Pk_Hnd_Store	Packet	Data

Figure 22. Packet_Handler Module Procedure Input/Output Parameters

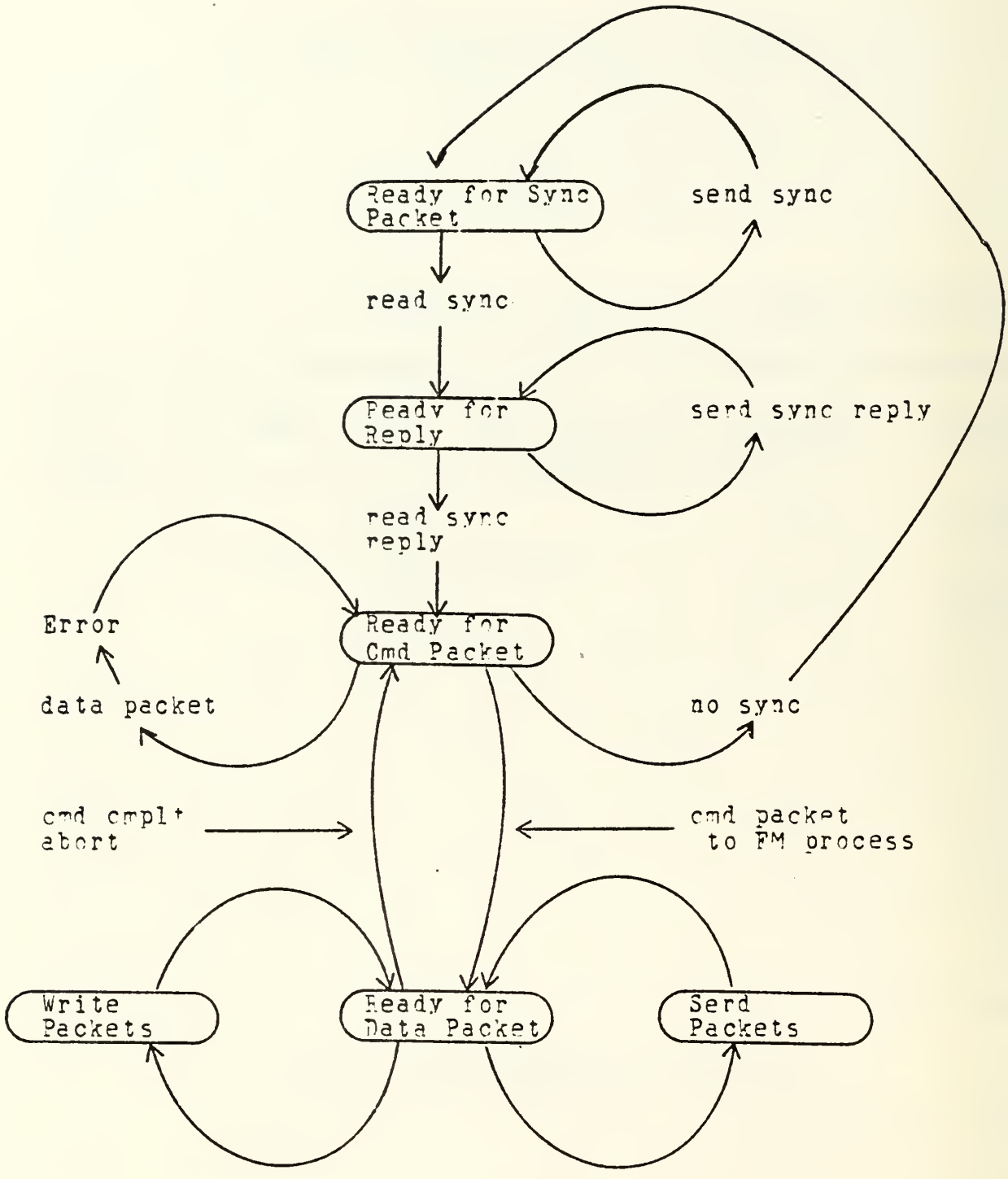


Figure 23. Finite State Diagram of Packet Transfer

synchronization procedure to begin synchronization in the middle of the first packet and still have two packets to confirm synchronization when it is achieved.

Packet transfer of command packets occurs one at a time. The reason for this is that each command packet must be acted upon in a synchronous manner. Data packet read ahead and write behind is permitted to increase the transfer rate of data packets. The number of packets that are allowed to be sent or stored depends on the IC buffer size. The Packet_Handler module is also responsible for data "enpacketing" and "depacketing" for the FSS.

The Pk_Hnd_Sync procedure is used to synchronize packet transmission. It is explicitly called at IPL and whenever the packet synchronization is lost by the Host system. It is invoked implicitly by the FSS whenever a packet is not able to be decoded (viz., the packet type and packet check-sum are incorrect).

The Pk_Hnd_Ack procedure is used to send acknowledgements to the Host systems. This procedure will always be called from the IO_Command_Handler module which will require the Packet_Handler module to either acknowledge the Host with a specific message or to send some data located in a mail_box segment buffer to the Host.

The Pk_Hnd_Send procedure is used to transfer data segments from the FSS to a Host system. This procedure is called from the File_Handler module which makes sure that the correct data segment is in process memory for the

trarsfer. The segment number along with the number of bits that are required to be transfered are passed to this procedure from the File_Handler module. This procedure then transfers the segment until the specified number of bits have been transmitted. A success code is returned when action is complete.

The Pk_Hnd_Store procedure works in a manner completely analogous to the Pk_Hnd_Read procedure.

III. CONCLUSIONS

A. STATUS OF RESEARCH

This design applies state of the art software and hardware to solve the secure multilevel computer problem in a file storage system. It presents an inexpensive but highly powerful design for a system based on a micro-computer but not restricted to a micro-computer environment, i.e., there is no restiction on the type of Host computer system serviced by the FSS. Implementation of this design on Z8000 hardware along with the analysis of FSS design parameters (Appendix A) are tasks left to be done.

There are two major classes of applications for the FSS. One application uses the FSS as a system file system (e.g., for distributed micros). This implies that the total system is multilevel secure with only one secure component (viz., the Kernel). It must be noted, however, that in this configuration, the distributed Hosts (i.e., the micros) have no autonomous life.

The other class of applications, involves using the FSS as one element of a net of autonomous Host systems. In this configuration, the FSS provides facilities for controlled data sharing and communication.

An obvious direct application of the FSS, is for shipboard use (e.g., for the SNAP-II system [Smith]) or for use at other installations where data would be more efficiently used if controlled data sharing were allowed.

A major design choice of the FSS which allowed the Kernel to be kept small (and therefore more easily verifiable), was the elimination of the discretionary security from the Kernel domain to the Supervisor domain. The implication of this choice is that each Host system is responsible for its own discretionary security; not an unreasonable request or design choice.

The next major task to be accomplished in this project is FSS implementation. This will not be a trival task, but it is felt that the designs presented in this thesis and the companion work done by Coleman provide a solid basis.

B. FOLLOW ON WORK

This design is a specific implementation of one member of a family of operating systems based on the Security Kernel concept discussed by O'Connell and Richardson [O'Connell]. There are obvious areas that this design could be expanded and generalized; areas that should be examined after a successful first implementation. Some of these areas are:

'operator' terminal interface functions

expanded Host commands

map of different 'user' names in different Hosts to a common "user" in the FSS

data compaction onto secondary storage

multilevel Hosts

moving discretionary security into the Kernel domain

dynamic process creation/deletion.

These are just a few of the many possible areas for expansion that could be explored. One area not mentioned in the list but an area that should be looked at during the initial implementation is for a way to prevent the Supervisor from suffering a 'segment fault'. The present arrangement, with a fault handler, is not efficient or 'elegant'. Since the deletion of a segment is controlled by the Delete_Segment Kernel primitive, a method of leaving an 'orphan' copy in process memory would eliminate the fault condition. The only operation that would be defined on this 'orphan' would be a Delete_Segment command by a process to remove it from process memory. After it had been deleted by all processes, the copy could be destroyed. A variation of this scheme would, upon a Kernel Swap_In call, swap into process memory a per-process copy of the desired segment. Swap_Out would be used to free process memory.

APPENDIX A--SYSTEM PARAMETERS

SMALL	Segment size	512 bytes
MEDIUM	Segment size	2K bytes
LARGE	Segment size	8K bytes
MAX_FILE_SIZE	Max file size	256K bytes
MAX_ENTRY	Max dir entries	32
ACL_POOL	Max Acl_Entries per directory	1024
Pathlength	Max	128 bytes
Entry_Name	Size	18 bytes
Known Segments	Max per process	--

APPENDIX B--SUCCESS AND ERROR CODES

CODE	LOCATION
File_Deleted	FM_Command_Handler
File_Created	Module
Link_Created	
Store_Complete	
Read_Complete	
ACL_Read_Complete	
ACL_Entry_Added	
ACL_Entry_Deleted	
Cmd_Aborted	
Cmd_Packet_Expected	
Illegal_Cmd	
Illegal_Cmd_Format	
File_Not_Found	Directory_Control
Not_Terminal_File	Module
Write_Access_Not_Allowed	Discretionary_Security
Read_Access_Not_Allowed	Module
Time_Out	Packet_Handler
No_Sync	Module
Packet_Ack	
Packet_Error	

```
FM_COMMAND_HANDLER MODULE

CONSTANT
  FALSE := 0
  TRUE := 1
  NULL := 0

EXTERNAL

DIR_CNTRL_DIRECTORY PROCEDURE (MSG           BYTE
                               USERID        BYTE
                               PATHNAME      STRING
                               FILE_TYPE     BYTE
                               ACCESS_LEVEL  BYTE
                               LINK          STRING
                               ACL_ENTRY     ACL_TYPE)
  RETURNS (DIR_SUCC_CODE                     BYTE)
!for host cmds that require parent directory:
 delete_file,
 create_file,
 create_link,
 read_acl,
 add_acl_entry,
 delete_acl_entry!

DIR_CNTRL_DATA PROCEDURE (MSG          BYTE
                          USERID       BYTE
                          PATHNAME     STRING)
                          FILE_SIZE    LWORD)
  RETURNS (DIR_SUCC_CODE               BYTE
           DIR_PATHNAME                STRING)
!for host cmds that access data file:
 read_file,
 store_file!

DIR_CNTRL_UPDATE PROCEDURE (MSG          BYTE
                            USERID       BYTE
                            PATHNAME     STRING)
  RETURNS (DIR_SUCC_CODE                 BYTE)
!to update directory after io process
 acts on host cmds: read_file, store_file,
 abort!
```

```
GLOBAL !module entry point!

FM_CMD_HND PROCEDURE !case statement on Host cmds!
ENTRY
DO
  MAIL_BOX.MSG.INST := READ_CMD
  MAIL_BOX.MSG.PATHNAME := NULL
  MAIL_BOX.MSG.FILE_SIZE := NULL
  MAIL_BOX.MSG.SUCC_CODE := NULL
  t := GATEKEEPER.TICKET (MAIL_BOX, C)
  GATEKEEPER.ADVANCE (MAIL_BOX, S)
  GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  IF MAIL_BOX.MSG.INST = CMD_PK_READY
    THEN
    IF HOST_CMD
      CASE DELETE_FILE THEN FM_CMD_HND_DELETE_FILE
      CASE CREATE_FILE THEN FM_CMD_HND_CREATE_FILE
      CASE CREATE_LINK THEN FM_CMD_HND_CREATE_LINK
      CASE READ_FILE THEN FM_CMD_HND_READ_FILE
      CASE STORE_FILE THEN FM_CMD_HND_STORE_FILE
      CASE READ_ACL THEN FM_CMD_HND_READ_ACL
      CASE ADD_ACL_ENTRY THEN FM_CMD_HND_ADD_ACL_ENTRY
      CASE DELETE_ACL_ENTRY THEN FM_CMD_HND_DELETE_ACL_ENTRY
      CASE ABORT THEN FM_CMD_HND_ABORT
    ELSE
      MAIL_BOX.MSG.INST := ACK_HOST
      MAIL_BOX.MSG.PATHNAME := NULL
      MAIL_BOX.MSG.FILE_SIZE := NULL
      MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (ILLEGAL_CMD)
      t := GATEKEEPER.TICKET (MAIL_BOX, C)
      GATEKEEPER.ADVANCE (MAIL_BOX, C)
      GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
    FI
  ELSE
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (CMD_PK_EXPECTED)
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  FI
OD
```

```
INTERNAL
MSG = BYTE

FM_CMD_HND_DELETE_FILE PROCEDURE
ENTRY
  MSG := DELETE_FILE
  DIR_CNTRL_DIRECTORY (MSG
                       USERID
                       PATHNAME
                       NULL !file_type!
                       NULL !access_level!
                       NULL !link!
                       NULL) !acl_entry!
  !returns dir_succ_code!
  IF DIR_SUCC_CODE = TRUE
    THEN
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := FILE_DELETED
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  ELSE
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
    !file not found; write access to directory
     not permitted!
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  FI
END FM_CMD_HND_DELETE_FILE
```

```
FM_CMD_HND_CREATE_FILE PROCEDURE
ENTRY
  MSG := CREATE_FILE
  DIR_CNTRL_DIRECTORY (MSG
                       USERID
                       PATHNAME
                       FILE_TYPE
                       ACCESS_LEVEL
                       NULL !link!
                       NULL) !acl_entry!
  !returns dir_succ_code!
  IF DIR_SUCC_CODE = TRUE
    THEN
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := FILE_CREATED
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  ELSE
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
    !directory not found; write access to directory
     not permitted; directory full!
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  FI
END FM_CMD_HND_CREATE_FILE
```

```
FM_CMD_CREATE_LINK PROCEDURE
ENTRY
  MSG := CREATE_LINK
  DIR_CNTRL_DIRECTORY (MSG
                       USERID
                       PATHNAME
                       NULL !file type!
                       NULL !access level!
                       LINK
                       NULL) !acl_entry!
  !returns dir_succ_code!
  IF DIR_SUCC_CODE = TRUE
    THEN
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := LINK_CREATED
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C (t+2))
  ELSE
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
    !directory not found; write access to directory
     not permitted; directory full!
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  FI
END FM_CMD_HND_CREATE_LINK
```

```
FM_CMD_READ_FILE PROCEDURE
ENTRY
  IF FILE_TYPE = DATA
    THEN
    MSG := READ_FILE
    DIR_CNTRL_DATA (MSG
                    USERID
                    PATHNAME
                    NULL) !file_size!
    !returns dir_succ_code, dir_pathname, dir_file_size!
    IF DIR_SUCC_CODE = TRUE
      THEN
      MAIL_BOX.MSG.INST := READ_FILE
      MAIL_BOX.MSG.PATHNAME := DIR_PATHNAME
      MAIL_BOX.MSG.FILE_SIZE := DIR_FILE_SIZE
      MAIL_BOX.MSG.SUCC_CODE := NULL
      t := GATEKEEPER.TICKET (MAIL_BOX, C)
      GATEKEEPER.ADVANCE (MAIL_BOX, C)
      GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
      IF MAIL_BOX.MSG.SUCC_CODE = TRUE
        THEN
        MSG := UPDATE_READ
        DIR_CNTRL_UPDATE (MSG
                          USERID
                          PATHNAME)
        !update will not fail!
        MAIL_BOX.MSG.INST := ACK_HOST
        MAIL_BOX.MSG.PATHNAME := NULL
        MAIL_BOX.MSG.FILE_SIZE := NULL
        MAIL_BOX.MSG.SUCC_CODE := READ_COMPLETE
        t := GATEKEEPER.TICKET (MAIL_BOX, C)
        GATEKEEPER.ADVANCE (MAIL_BOX, C)
        GATEKEEPER.AWAIT (MAIL_BOX, C)
      ELSE
        MAIL_BOX.MSG.INST := ACK_HOST
        MAIL_BOX.MSG.PATHNAME := NULL
        MAIL_BOX.MSG.FILE_SIZE := NULL
        MAIL_BOX.MSG.SUCC_CODE := MAIL_BOX.MSG.SUCC_CODE
        !error code returned from io process!
        !file not found by io process;
         file read aborted by  write;
         file read aborted by file deletion;
         cmd packet received!
        t := GATEKEEPER.TICKET (MAIL_BOX, C)
        GATEKEEPER.ADVANCE (MAIL_BOX, C)
        GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
      FI
    ELSE
      MAIL_BOX.MSG.INST := ACK_HOST
      MAIL_BOX.MSG.PATHNAME := NULL
      MAIL_BOX.MSG.FILE_SIZE := NULL
      MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
      !file not found;
       read access to file not permitted!
```

```
      t := GATEKEEPER.TICKET (MAIL_BOX, C)
      GATEKEEPER.ADVANCE (MAIL_BOX, C)
      GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
    FI
  ELSE
    IF FILE_TYPE = DIRECTORY
      THEN
      MSG := READ_DIR
      DIR_CNTRL_DIRECTORY (MSG
                           USERID
                           PATHNAME)
                           NULL !file_type!
                           NULL !access_level!
                           NULL !link!
                           NULL) !acl_entry!
      !returns dir_succ_code!
      IF DIR_SUCC_CODE = TRUE
        THEN
        MAIL_BOX.MSG.INST := ACK_HOST
        MAIL_BOX.MSG.PATHNAME := NULL
        MAIL_BOX.MSG.FILE_SIZE := NULL
        MAIL_BOX.MSG.SUCC_CODE := DIR_READ_COMPLETE
        !dir data transfered from dir_buffer;
         acknowledgement sent!
        t := GATEKEEPER.TICKET (MAIL_BOX, C)
        GATEKEEPER.ADVANCE (MAIL_BOX, C)
        GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
      ELSE
        MAIL_BOX.MSG.INST := ACK_HOST
        MAIL_BOX.MSG.PATHNAME := NULL
        MAIL_BOX.MSG.FILE_SIZE := NULL
        MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
        !directory not found,
         read access to directory not permitted!
        t := GATEKEEPER.TICKET (MAIL_BOX, C)
        GATEKEEPER.ADVANCE (MAIL_BOX, C)
        GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
      FI
    ELSE
      MSG := READ_ENTRY_DATA
      DIR_CNTRL_DIRECTORY (MSG
                           USERID
                           PATHNAME
                           NULL !file_type!
                           NULL !access_level!
                           NULL !link!
                           NULL) !acl_entry!
      !returns dir_succ_code!
      IF DIR_SUCC_CODE = TRUE
        THEN
        MAIL_BOX.MSG.INST := ACK_HOST
        MAIL_BOX.MSG.PATHNAME := NULL
        MAIL_BOX.MSG.FILE_SIZE := NULL
        MAIL_BOX.MSG.SUCC_CODE := ENTRY_READ_COMPLETE
```

```
        !entry data transfered from dir_buffer;
         acknowledgement sent!
        t := GATEKEEPER.TICKET (MAIL_BOX, C)
        GATEKEEPER.ADVANCE (MAIL_BOX, C)
        GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
      ELSE
        MAIL_BOX.MSG.INST := ACK_HOST
        MAIL_BOX.MSG.PATHNAME := NULL
        MAIL_BOX.MSG.FILE_SIZE := NULL
        MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
        !file not found; read access to file not permitted!
        t := GATEKEEPER.TICKET (MAIL_BOX, C)
        GATEKEEPER.ADVANCE (MAIL_BOX, C)
        GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
      FI
    FI
  FI
END FM_CMD_HND_READ_FILE
```

```
FM_CMD_HND_STORE_FILE PROCEDURE
ENTRY
  MSG := STORE_FILE
  DIR_CNTRL_DATA  (MSG
                   USERID
                   PATHNAME
                   FILE_SIZE)
  !returns dir_pathname; dir_succ_code!
  IF DIR_SUCC_CODE = TRUE
    THEN
    MAIL_BOX.MSG.INST := STORE_FILE
    MAIL_BOX.MSG.PATHNAME := DIR_PATHNAME
    MAIL_BOX.MSG.FILE_SIZE := FILE_SIZE
    MAIL_BOX.MSG.SUCC_CODE := NULL
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
    IF MAIL_BOX.MSG.SUCC_CODE = TRUE
      THEN
      MSG := UPDATE_STORE
      DIR_CNTRL_UPDATE (MSG
                        USERID
                        PATHNAME)
      !update will not fail!
      MAIL_BOX.MSG.INST := ACK_HOST
      MAIL_BOX.MSG.PATHNAME := NULL
      MAIL_BOX.MSG.FILE_SIZE := NULL
      MAIL_BOX.MSG.SUCC_CODE := STORE_COMPLETE
      t := GATEKEEPER.TICKET (MAIL_BOX, C)
      GATEKEEPER.ADVANCE (MAIL_BOX, C)
      GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
    ELSE
      MAIL_BOX.MSG.INST := ACK_HOST
      MAIL_BOX.MSG.PATHNAME := NULL
      MAIL_BOX.MSG.FILE_SIZE := NULL
      MAIL_BOX.MSG.SUCC_CODE := MAIL_BOX.MSG.SUCC_CODE
      !error returned from io process;
       cmd packet received: improper number of data packets!
      t := GATEKEEPER.TICKET (MAIL_BOX, C)
      GATEKEEPER.ADVANCE (MAIL_BOX, C)
      GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
    FI
  ELSE
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
    !file not found; write access to file not permitted!
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  FI
END FM_CMD_HND_STORE_FILE
```

```
FM_CMD_HND_READ_ACL PROCEDURE
ENTRY
  MSG := READ_ACL
  DIR_CNTRL_DIRECTORY (MSG
                       USERID
                       PATHNAME
                       NULL !file type!
                       NULL !access level!
                       NULL !link!
                       NULL) !acl_entry!
  !returns dir_succ_code!
  IF DIR_SUCC_CODE = TRUE
    THEN
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ACL_READ_COMPLETE
    !acl data transfered from acl_buffer;
     host acknowledgement sent!
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  ELSE
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
    !file not found; read access to directory file
     not permitted!
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  FI
END FM_CMD_HND_READ_ACL
```

```
FM_CMD_END_ADD_ACL_ENTRY PROCEDURE
ENTRY
  MSG := ADD_ACL_ENTRY
  DIR_CNTRL_DIRECTORY (MSG
                       USERID
                       PATHNAME
                       NULL !file type!
                       NULL !access level!
                       NULL !link!
                       ACL_ENTRY)
  !returns dir_succ_code!
  IF DIR_SUCC_CODE = TRUE
    THEN
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ACL_ENTRY_ADDED
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  ELSE
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
    !file not found; write access to directory not
     permitted; acl_entry "pool" empty'
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  FI
END FM_CMD_END_ADD_ACL_ENTRY
```

```
FM_CMD_HND_DELETE_ACL_ENTRY PROCEDURE
ENTRY
  MSG := DELETE_ACL_ENTRY
  DIR_CNTRL_DIRECTORY (MSG
                       USERID
                       PATHNAME
                       NULL !file_type!
                       NULL !access_level!
                       NULL !link!
                       ACL_ENTRY)
  !returns dir_succ_code!
  IF DIR_SUCC_CODE = TRUE
    THEN
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ACL_ENTRY_DELETED
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  ELSE
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
    !file not found; write access to directory not
     permitted!
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  FI
END FM_CMD_HND_DELETE_ACL_ENTRY
```

```
FM_CMD_HND_APORT PROCEDURE
ENTRY
  MSG := APORT
  DIR_CNTRL_UPDATE (MSG
                    USERID
                    PATHNAME)
  !store cmd needs to free temporary file!
  MAIL_BOX.MSG.INST := ACK_HOST
  MAIL_BOX.MSG.PATHNAME := NULL
  MAIL_BOX.MSG.FILE_SIZE := NULL
  MAIL_BOX.MSG.SUCC_CODE := CMD_ABORTED
  t := GATEKEEPER.TICKET (MAIL_BOX, C)
  GATEKEEPER.ADVANCE (MAIL_BOX, C)
  GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
END FM_CMD_HND_ABORT

END FM_COMMAND_HANDLER
```

```
IO_COMMAND_HANDLER MODULE

EXTERNAL


PK_HND_STORE PROCEDURE (SEG_#      LWORD
                        SIZE       LWORD) !number of bits!
  RETURNS (PK_SUCC_CODE            BYTE)

PK_HND_SEND PROCEDURE (SEG_#       LWORD
                       SIZE        LWORD) !number of bits!
  RETURNS (PK_SUCC_CODE            BYTE)

PK_HND_ACK_HOST PROCEDURE (MSG     BYTE)

FILE_HND_SEND_FILE PROCEDURE (PATHNAME      STRING
  RETURNS (FILE_SUCC_CODE                   BYTE)

FILE_HND_STORE_FILE PROCEDURE (PATHNAME     STRING)
  RETURNS (FILE_SUCC_CODE                   BYTE)

INTERNAL

IO_CMD_HND PROCEDURE
ENTRY
  t := TICKET (MAIL_BOX, C)
  AWAIT (MAIL_BOX, C, (t+1))
  DO
    IF MAIL_BOX.MSG.INST
      CASE READ_CMD THEN PK_HND_READ_CMD
      CASE ACK_HOST THEN PK_HND_ACK_HOST
                         (MAIL_BOX.MSG.SUCC_CODE)
      CASE SEND_FILE THEN FILE_HND_READ_FILE
                         (MAIL_BOX.MSG.PATHNAME
                         (MAIL_BOX.MSG.FILE_SIZE)
      CASE STORE_FILE THEN FILE_HND_STORE_FILE
                           (MAIL_BOX.MSG.PATHNAME
                            MAIL_BOX.MSG.FILE_SIZE)
    FI
    t := TICKET (MAIL_BOX, C)
    ADVANCE (MAIL_BOX, C)
    AWAIT (MAIL_BOX, C, (t+2)
  OD
END IO_CMD_HND

END IO_COMMAND_HANDLER
```

LIST OF REFERENCES

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

COURTOIS, P. J., Heymans, F., and Parnas, D. L., "Concurrent Control with "Readers" and "Writers"," Communications of the ACM, v.14 no.5 p.667-668, October 1971.

DAVIES, D. W., and others, Computer Networks and Their Protocols, John Wiley & Sons, 1979.

DEFENSE Communications Agency NIC 7104, Arpanet Protocol Handbook, by Network Information Center, January 1978.

DENNING(1), D. E., "A Lattice Model of Secure Information Flow," Communications of the ACM, v. 19 p. 236-242, May 1976.

DENNING(2), D. E. and Denning, P. J., "Data Security," ACM Computing Surveys, v. 11 no. 3, p. 227-242, May 1976.

DIGITAL Research, CP/M Interface Guide, 1978.

DIJKSTRA(1), E. W., "The Structure of 'The' Multiprogramming System," Communications of the ACM, v. 11 no. 5, p. 341-346, May 1968.

DIJKSTRA(2), E. W., "The Humble Programmer," Communications of the ACM, v. 15 no. 10, p. 859-866, October 1972.

HABERMANN, A. N., Introduction to Operating System Design, Science Research Associates, Inc., 1976.

HAMMING, D. E., Coding and Information Theory, Prentice Hall, Inc., 1980.

HANSON, B., Operating System Design, Printice-Hall, Inc., 1973.

HONEYWELL, The Multics Virtual Memory, June 1972.

MADNICK, S. E. and Donovan, J. J., Operating Systems, McGraw Hill, 1974.

MORRIS, R. and Thompson, K., Password Security, A Case History, paper presented to Bell Labratories, April 3, 1978.

MITRE Corporation Report 2934, The Design and Specification of a Security Kernel for the PDP-11/45, by W. L. Schiller, May 1975.

O'CONNELL, J. S., and Richardson, L. D., Distributed Secure

Design for a Multi-microprocessor Operating System, MS Thesis, Naval Postgraduate School, June 1979.

ORGANICK, E. I., The Multics System: An Examination of Its Structure, MIT Press, 1972.

REED, P. D., and Kanodia, R. K., "Synchronization with Eventcounts and Sequencers," Communications of the ACM, v. 22 no. 2 p. 115-124, February 1979.

SMITH, D. L., Method to Evaluate Microcomputers for Non-Tactical Shipboard Use, MS Thesis, Naval Postgraduate School, September 1979.

SCHELL(1), Lt.Col. R. R., "Computer Security: The Achilles' Heel of the Electronic Air Force?," Air University Review, v.XXX no. 2, January 1979.

SCHELL(2), Lt.Col. R. R., Security Kernels: A Methodical Design of System Security, USE Technical Papers (Spring Conference, 1979). pp 245-257, March 1979.

SCHROEDER, M. D., "A Hardware Architecture for Implementing Protection Rings," Communications of the ACM, v. 15 no. 3, p. 157-170, March 1972.

SHAW, A. C., The Logical Design of Operating Systems, Prentice Hall, Inc., 1974.

SNOOK, T., and others, Report on the Programming Language PLZ/SYS, Springer-Virlag, 1978.

ZILOG(1), Inc., An introduction to the Z8010 MMU Memory Management Unit, Tutorial Information, August 1979.

ZILOG(2), Inc., Z8001 CPU, Priliminary Product Specification, March 1979.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Office of Research Administration Code 012A Naval Postgraduate School Monterey, California 93940	1
4.	Chairman, Code 52Bz Computer Science Department Naval Postgraduate School Monterey, California 93940	4
5.	LTCOL Roger R. Schell, Code 52Sj Computer Science Department Naval Postgraduate School Monterey, California 93940	15
6.	Lyle A. Cox, Jr., Code 52Cl Computer Science Department Naval Postgraduate School Monterey, California 93940	5
7.	Uno R. Kodres, Code 52Kr Computer Science Department Naval Postgraduate School Monterey, California 93940	2
8.	I. Larry Avrunin, Code 18 DTNSRDC Bethesda, MD 20084	1
9.	R. P. Crabb, Code 9134 Naval Ocean Systems Center San Diego, CA 92152	1
10.	G. H. Gleissner, Code 18 DTNSRDC Bethesda, MD 20084	1
11.	Kathryn Heninger, Code 7503 Naval Research Lab Washington, D.C. 20375	1

12. Ronald P. Kasik, Code 4451 — 1
Naval Underwater Systems Center
Newport, RI 02840

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

14. George Mebus, Code 5033 — 1
NADC
Warminster, PA 18974

15. Joel Trimble, Code 221 — 1
Office of Naval Research
800 North Quincy
Arlington, Virginia 22217

16. Mark Underwood, Code P204 — 1
NPRDC
San Diego, CA 92152

17. Michael Wallace, Code 1828 — 1
DTNSRDC
Bethesda, MD 20084

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

19. John Zenor, Code 31302 — 1
Naval Weapons Center
China Lake, CA 93555

20. Chief of Naval Research — 1
Arlington, Virginia 22217