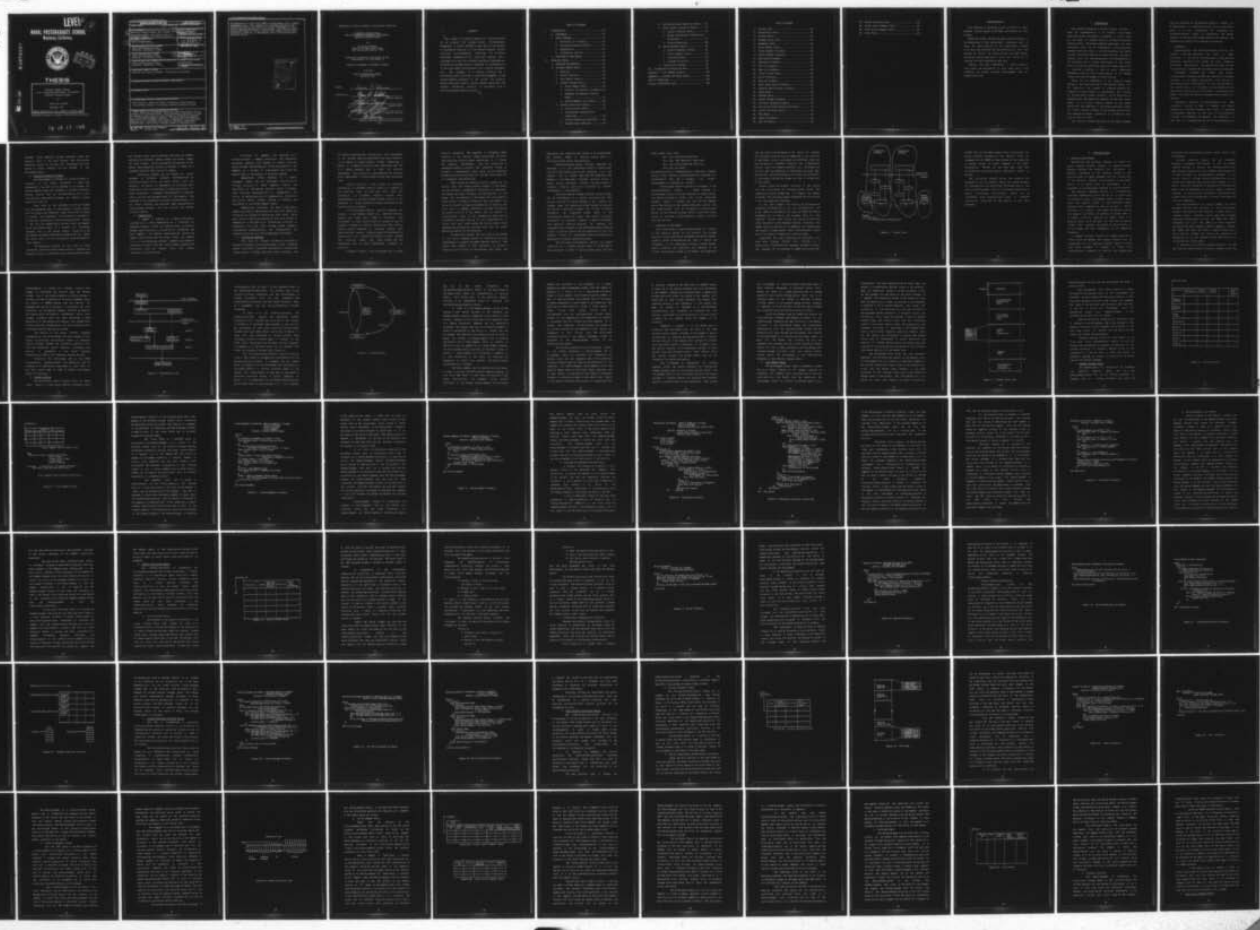
AD-A078 281 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
SECURITY KERNEL DESIGN FOR A MICRO-PROCESSOR-BASED, MULTILEVEL --ETC(U)
DEC 79 A R COLEMAN

UNCLASSIFIED NL

1 OF 2
AD A078281



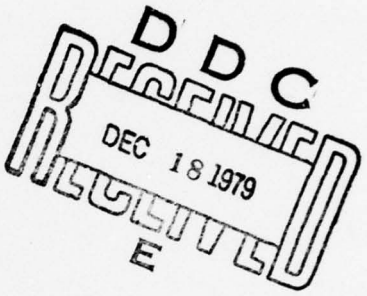
LEVEL II

2

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AD A078281



THESIS

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

by

Aaron Ray Coleman

December 1979

Thesis Advisor: Lt. Col. Roger R. Schell,USAF

Approved for public release: distribution unlimited

DDC FILE COPY

79 12 17 090

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

(6) Security Kernel Design for a Microprocessor-Based, Multilevel Archival Storage System.

(9) 5. TYPE OF REPORT & PERIOD COVERED
Master's Thesis, December, 1979

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

(10) Aaron Ray Coleman

8. CONTRACT OR GRANT NUMBER(s)
(12) 109

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Naval Postgraduate School
Monterey, California 93940

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

11. CONTROLLING OFFICE NAME AND ADDRESS
Naval Postgraduate School
Monterey, California 93940

(11) 12. REPORT DATE
Dece[illegible]79

13. NUMBER OF PAGES
108

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)
Naval Postgraduate School
Monterey, California 93940

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)
Data Security, Security Kernel, Operating System Security
File System, Secure, File System, Secure Operating System

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
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 controled sharing, and a loop-free structure for avoiding intermodule dependencies, the Archival Storage System is designed for

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EDITION OF 1 NOV 65 IS OBSOLETE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

251 450 gm

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

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.

Accession For
NTIS GRA&I ☑
DDC TAB ☐
Unannounced ☐
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or special
A

Approved for public release; distribution unlimited.

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

Author Aaron R Coleman

Approved by: Roger R. Schell
Thesis Advisor

Second Reader

Chairman, Department of Computer Science

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

I. INTRODUCTION.....10

A. BACKGROUND.....11

B. BASIC CONCEPTS.....12

1. Definition of a Process.....13

2. Multiple Protection Domains.....14

3. Segmentation.....15

4. Information Security.....16

C. STRUCTURE OF THE THESIS.....20

II. DETAILED DESIGN.....24

A. HARDWARE REQUIREMENTS.....24

B. PROPOSED KERNEL DESIGN.....27

1. Notation.....27

2. Kernel Overview.....28

3. Gate Keeper Module.....35

4. Segment Manager Module.....38

a. Known Segment Table.....40

b. Creation and Deletion of Segments.41

c. Managing the Segmented Address Space.....47

d. Moving Segments into Memory.....53

5. Traffic Controller Module.....57

a. Active Process Table.....57

b. Interprocess Communication Primitives.....61

c. Process Scheduling Algorithm.....65

d. Message Queue Operators.....69

6. Non-Discretionary Security Module.....71
7. Inner Traffic Controller Module.....75
a. Virtual Processor Table.....76
b. Kernel Interprocess Communication Primitives.....76
c. Service Functions.....82
8. Memory Manager Module.....83
a. Memory Management Scheme.....84
b. Active Segment Table.....87
c. Aliasing Scheme.....92
d. Storage Allocation.....94
9. Input-Output Manager.....95
III. CONCLUSION AND FOLLOW ON WORK.....97
APPENDIX A - GATE KEEPER LISTING.....100
APPENDIX B - SUCCESS AND ERROR CODES.....104
LIST OF REFERENCES.....106
INITIAL DISTRIBUTION LIST.....108

LIST OF FIGURES

1. Process View.....22
2. Hierarchical View.....29
3. Process States.....31
4. Program Status Area.....37
5. Parameter Table.....39
6. Known Segment Table.....42
7. Create Segment Procedure.....44
8. Delete Segment Procedure.....46
9. Make Known Procedure.....48
10. Terminate Procedure.....52
11. Swap In Procedure.....54
12. Swap Out Procedure.....55
13. Active Process Table.....58
14. Block Procedure.....62
15. Wake Up Procedure.....64
16. Enter Ready Queue Procedure.....66
17. Schedule Ready Process Procedure.....67
18. Ready Queue.....68
19. Message Queue.....70
20. Insert Message Procedure.....72
21. Get First Message Procedure.....73
22. Non-Discretionary Security Procedure.....74
23. Virtual Processor Table.....77
24. MMU Image.....78
25. Signal Procedure.....80
26. Wait Procedure.....81

27. Memory Allocation Map.....86
28. Global Active Segment Table.....88
29. Local Active Segment Table.....88
30. Alias Table.....93

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.

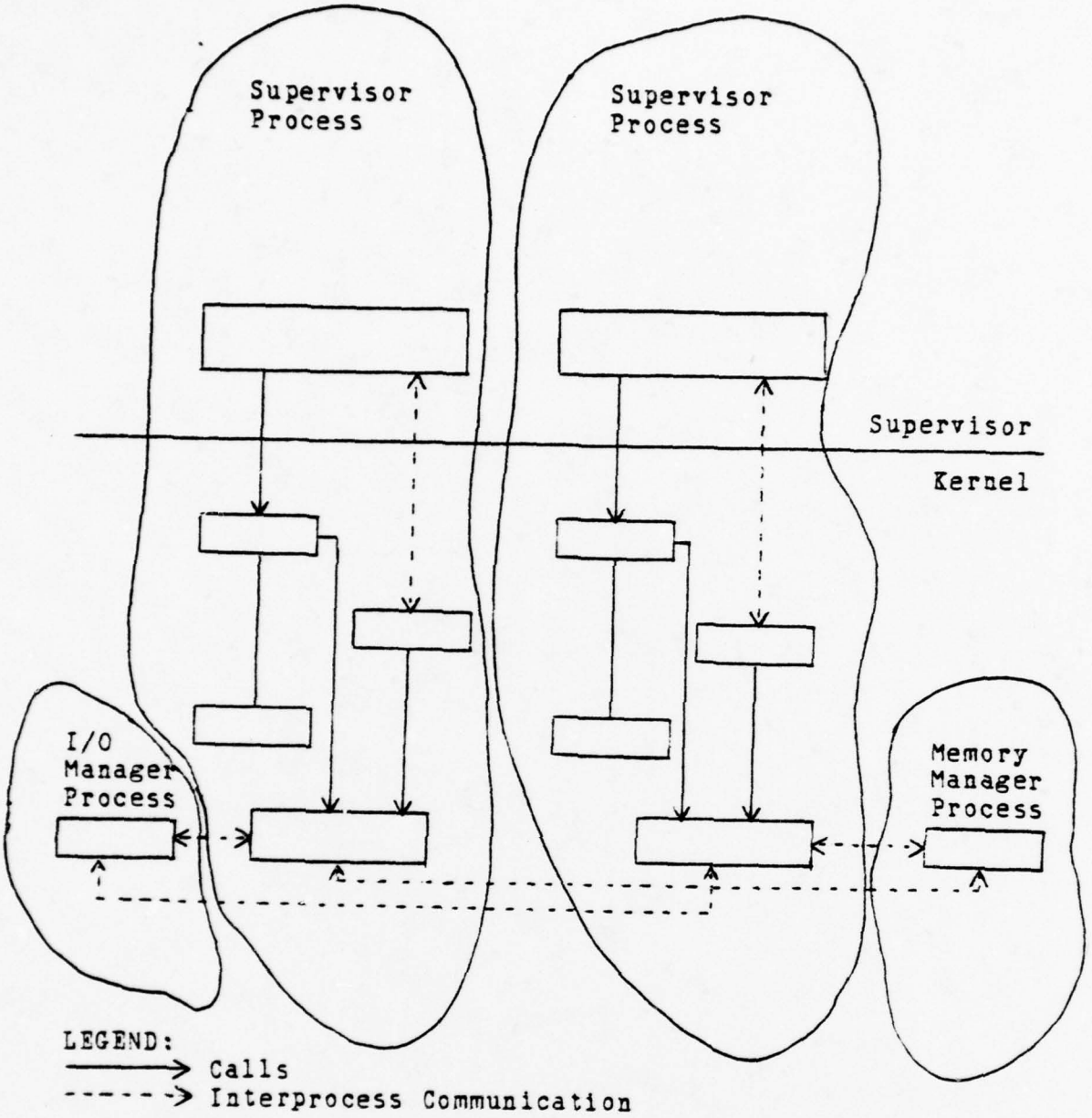


Figure 1. Process View

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

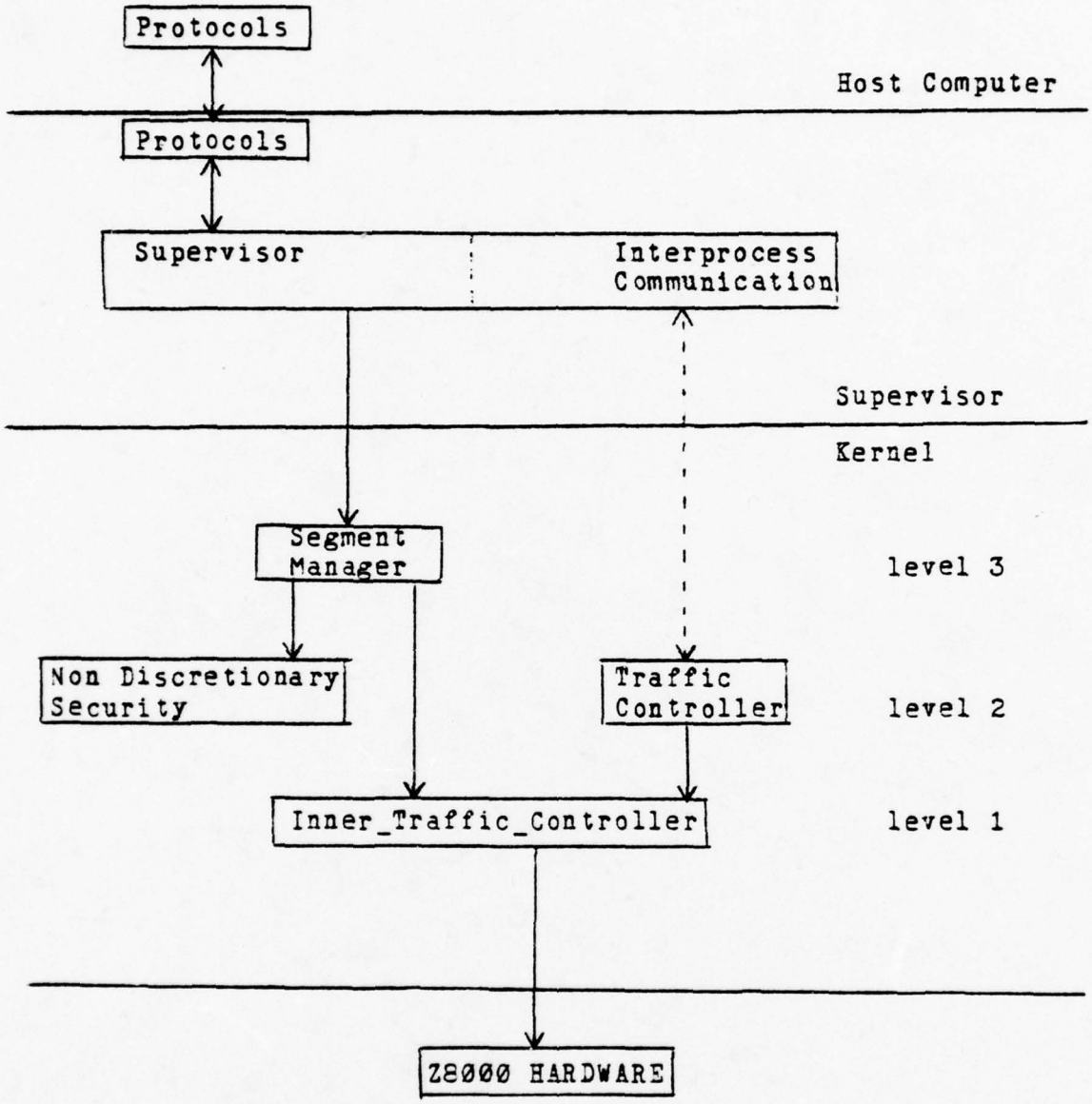


Figure 2. Hierarchical View

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)

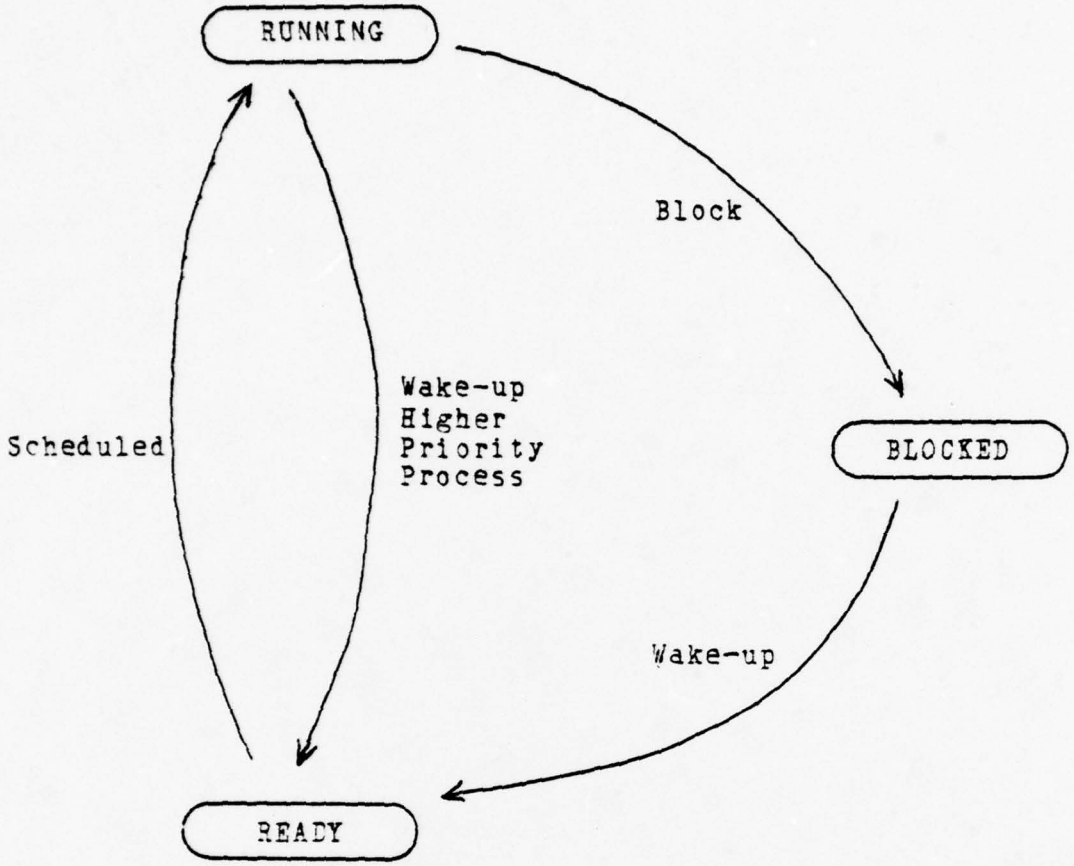


Figure 3. Process States

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!)

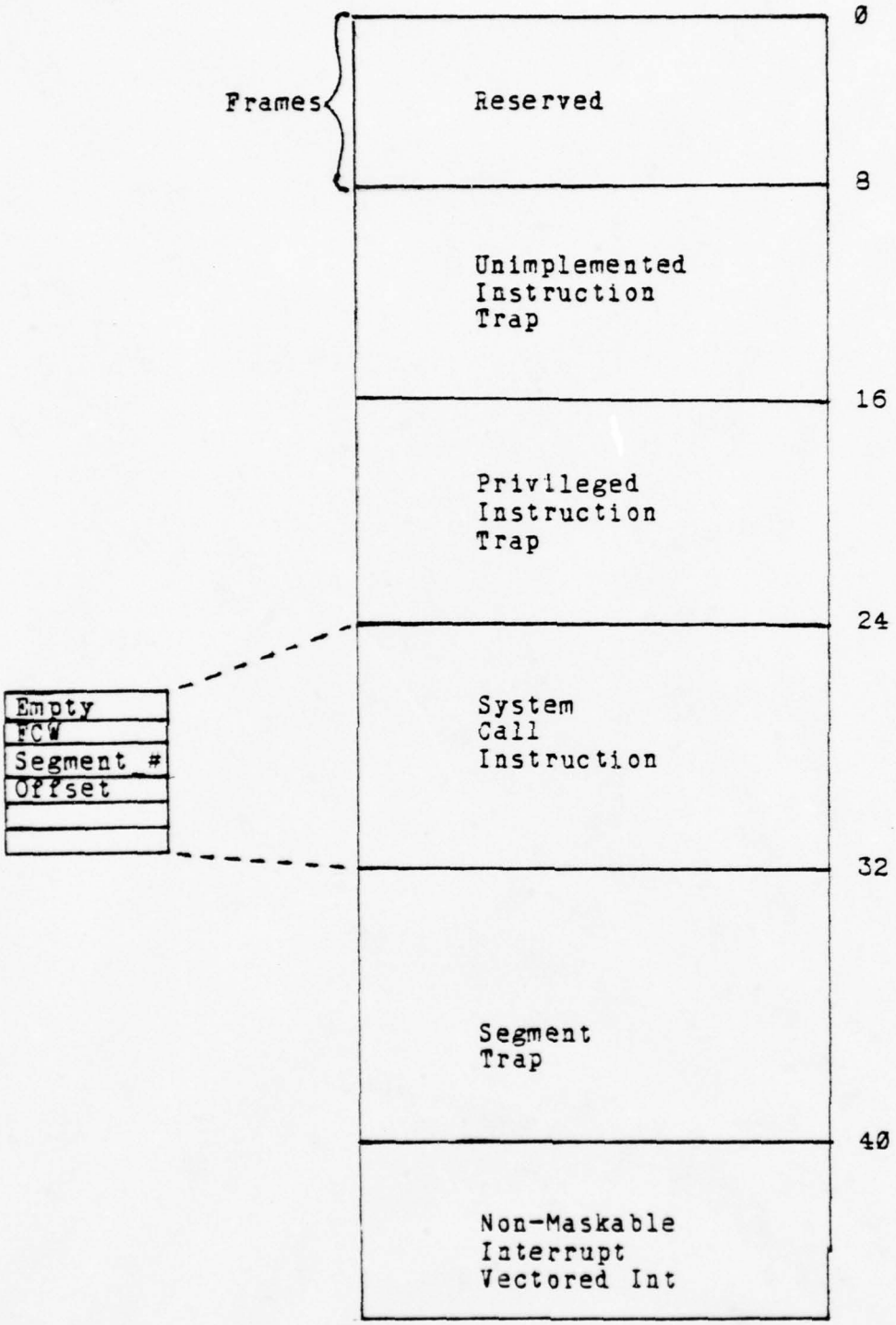


Figure 4. Program Status Area

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.

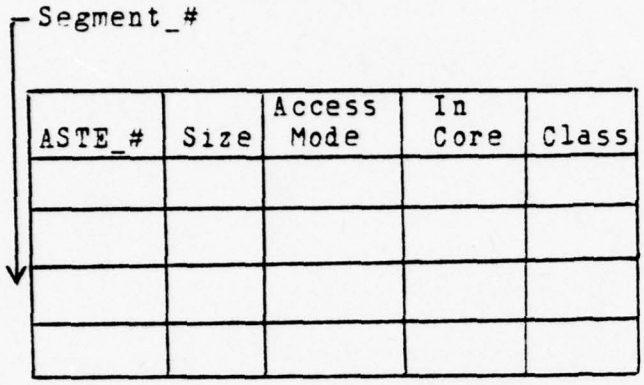
Figure 6 shows both the logical view and the PLZ variable declaration for the KST. Max_KST_Size is hardware dependent and is equal to the maximum number of segments which can be mapped by the MMU. To access an element of the database the following notation is used:

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.



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 := Ø
            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

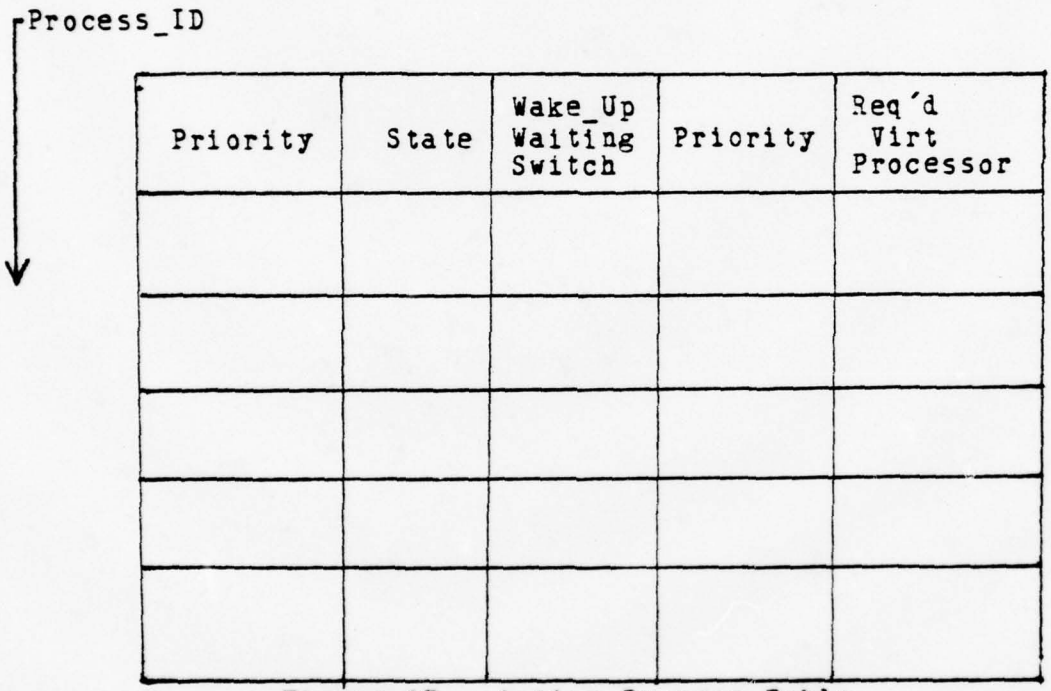


Figure 13. Active Process Table

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] := Ø
  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

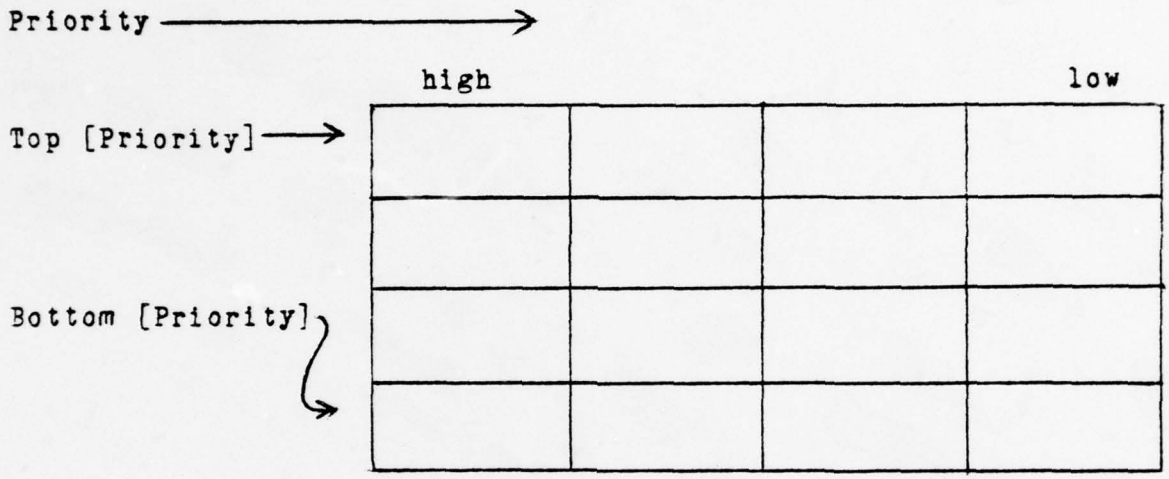


Figure 18. Ready Queue

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

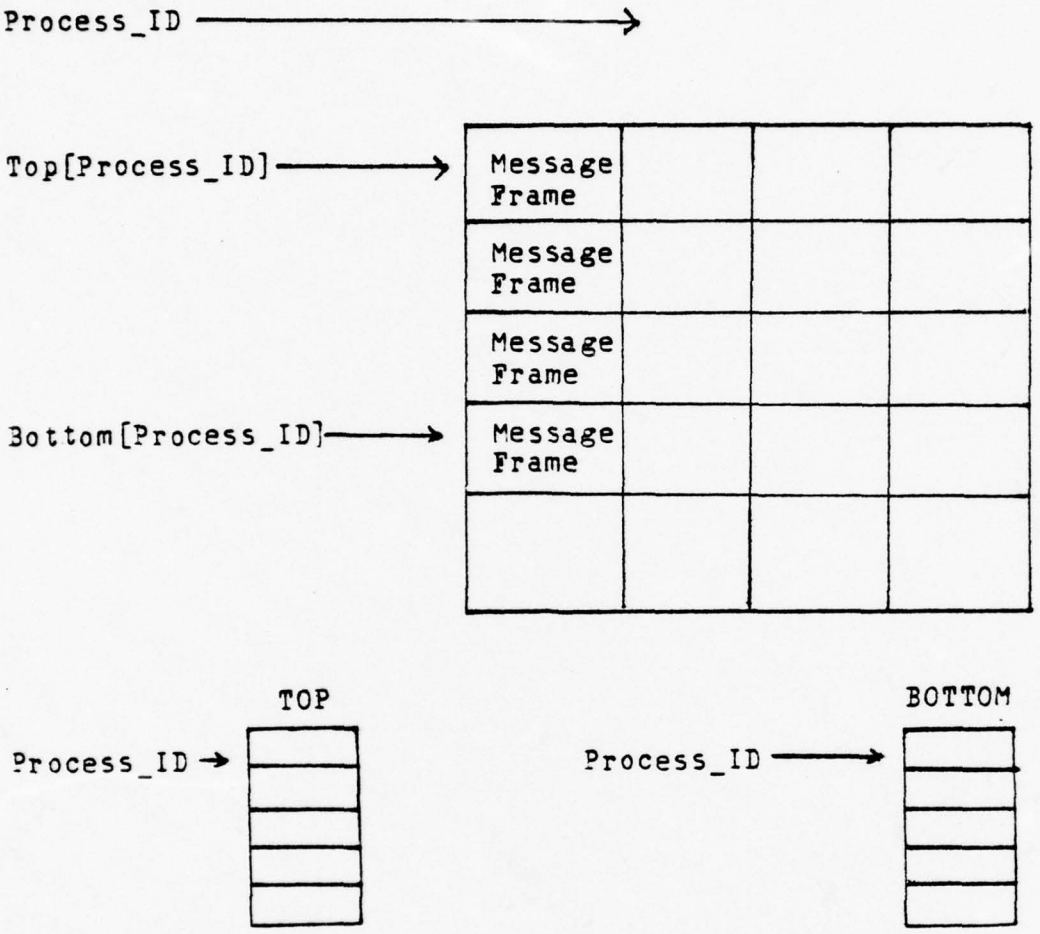


Figure 19. Message Queue And Pointers

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

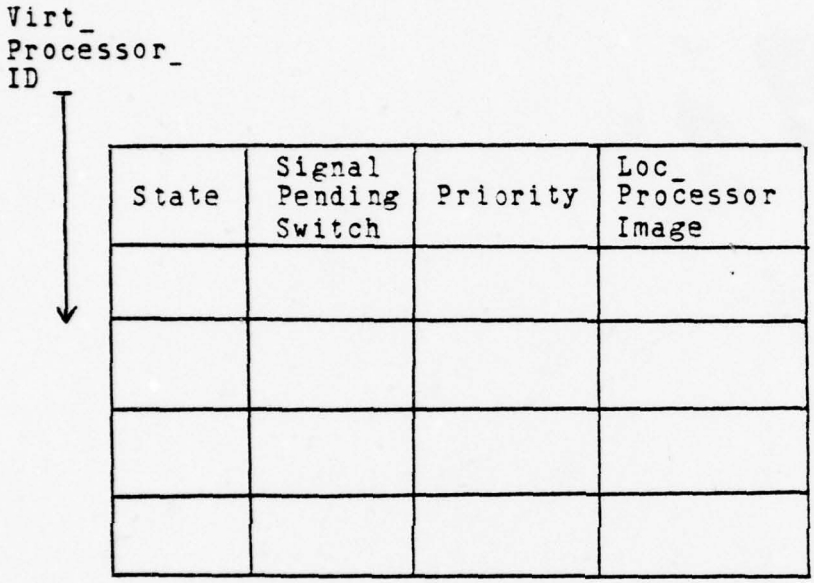


Figure 23. Virtual Processor Table

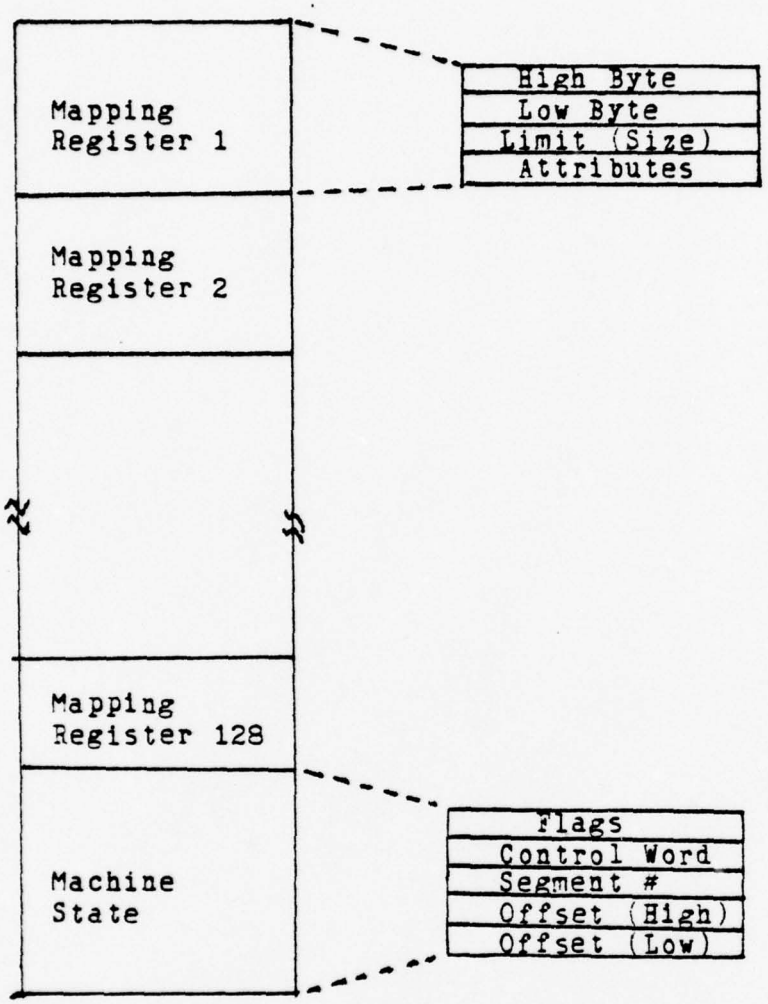


Figure 24. MMU Image

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 Message)
    If Success_Code = Queue_Overflow
    Orif Success_Code = Not_Allowed
    Then Exit
    Else VPT [Signaled_Processor_ID].Signal_Pending_Switch := On
         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

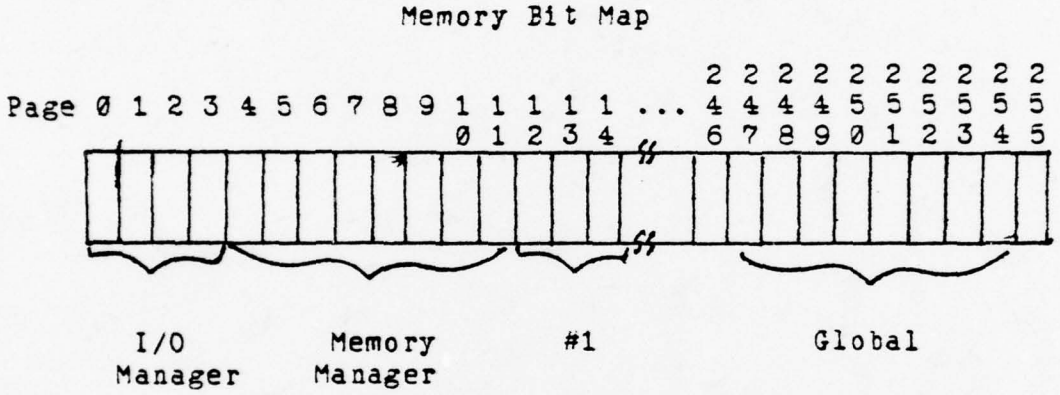


Figure 27. Memory Allocation Map

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

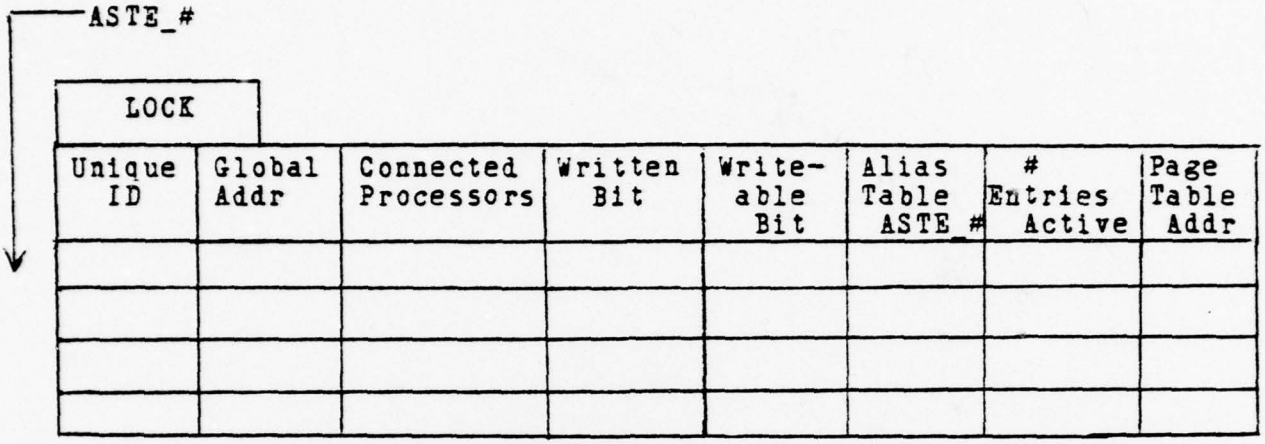


Figure 28. Global Active Segment Table

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

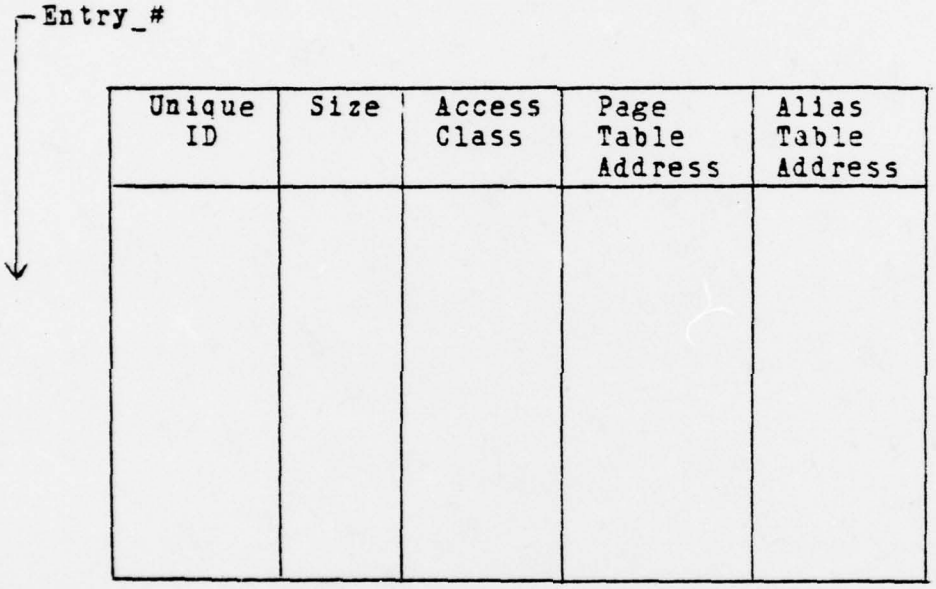


Figure 30. Alias Table

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

AD-A078 281 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
SECURITY KERNEL DESIGN FOR A MICRO-PROCESSOR-BASED, MULTILEVEL --ETC(U)
DEC 79 A R COLEMAN

UNCLASSIFIED NL

2 OF 2
AD
A078281

END
DATE
FILMED
1-80
DDC

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.

APPENDIX A - GATE KEEPER LISTING

```
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,RH0
            IF NE THEN EXIT FROM VALIDATE  !Return if invalid!
            ELSE LDL RR2,<<DIST_KERNEL_ID>>ARGUMENT_POINTER
                 LDB RH0,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.

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.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	Lt.Col. R. R. Schell, Code 52Sj Department of Computer Science Naval Postgraduate School Monterey, California 93940	5
5.	Asst Professor Lyle A. Cox, Code 52Cl Department of Computer Science Naval Postgraduate School Monterey, California 93940	3
6.	Mr. Joel Trimble, Code 221 Office of Naval Research 800 North Quincy Arlington, Virginia 22217	1
7.	CPT A. R. Coleman Box 426 U.S. Army War College Carlisle, Pennsylvania 17013	2
8.	Lt. E. J. Parks SMC 1910 Naval Postgraduate School Monterey, California 93940	1