PHOTOGRAPH THIS SHEET

AD A087975

DTIC ACCESSION NUMBER

II

LEVEL

INVENTORY

Naval Postgraduate School
Monterey, CA
"The Design of a Secure File Storage System"
Master's Thesis Dec. 1979 Edw. James Parks

DOCUMENT IDENTIFICATION

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DISTRIBUTION STATEMENT

ACCESSION FOR	
NTIS GRA&I	☒
DTIC TAB	☐
UNANNOUNCED	☐
JUSTIFICATION	
BY	
DISTRIBUTION /	
AVAILABILITY CODES	
DIST	AVAIL AND/OR SPECIAL
A	23 cy

DISTRIBUTION STAMP

DTIC
ELECTE
AUG 18 1980
S D
D

DATE ACCESSIONED

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DDC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

DATE RECEIVED IN DTIC

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDA-2

DTIC FORM OCT 79 70A

DOCUMENT PROCESSING SHEET

NAVAL POSTGRADUATE SCHOOL
Monterey, California

DEPARTMENT OF THE NAVY
NAVAL POSTGRADUATE SCHOOL

THESIS

The Design of a Secure File Storage System

by

Edward James Parks

December 1979

Thesis Advisor: L.A. Cox

Approved for public release; distribution unlimited

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Design of a Secure File Storage System		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis;Dec 79
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Edward James Parks		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE December 1979
		13. NUMBER OF PAGES 120
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 Systems, File System, Secure File System, Secure Operating System		

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

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

THE DESIGN OF A SECURE FILE STORAGE SYSTEM

by

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

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1979

Author [illegible]

Approved by: [illegible]
Thesis Advisor

Roger R. Schell
Second Reader

[illegible]
Chairman, Department of Computer Science

[illegible]
Dean of Information and Policy Sciences

ABSTRACT

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

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENT

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

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

I. INTRODUCTION

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

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

In contrast. this thesis presents a design for a multilevel secure computer operating system. the File Storage System (FSS) in which internal computer security is a primary design objective. There are two goals this system is designed to achieve: 1) to provide sharing of data among authorized users and. 2) control access to a consolidated "warehouse" of data. This controlled access to consolidated data, predicates a "star" network for the system structure as depicted in figure 1. It must be noted, however, that the FSS cannot control the physical security of the Host systems and that Host systems have the ability to circumvent FSS security by direct inter-Host communication links. To preserve data security, all accesses to the FSS consolidated data must go through the FSS for access validation.

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

A. PROBLEM DEFINITION

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

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

A system that can simultaneously provide data at

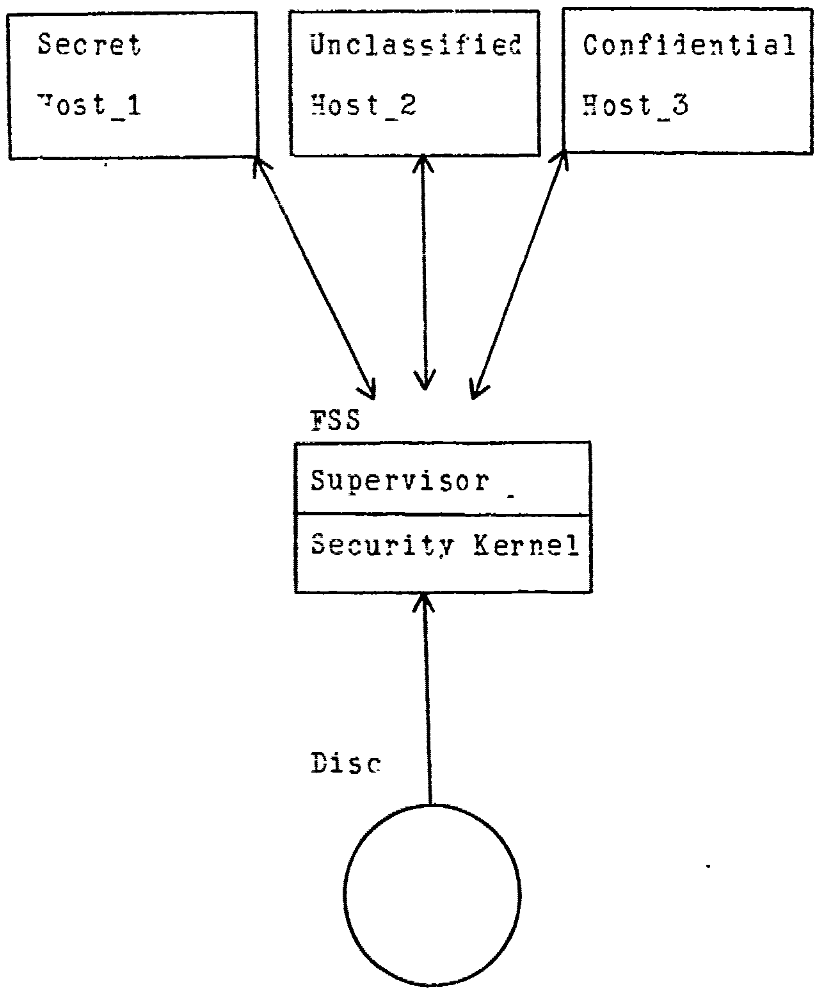


Figure 1. System Configuration

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

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

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

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

B. BACKGROUND

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

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

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

Pay and Personnel

Supply and Finance

Maintenance.

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

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

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

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

C. BASIC DEFINITIONS

1. Security

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

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

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

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

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

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

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

. implement its aspect of security, the Supervisor needs to know the identification of the Host system "user". This Host system user identification must be passed to the FSS Supervisor by the Host system. Since an insecure Host system cannot be trusted to pass the correct information, the user identification is only as good as the Host system implementation. (i.e., FSS discretionary security is only as good as the Host System's implementation of discretionary security.) This implementation may be good on some systems, (e.g., UNIX [Morris]) but non-existent on other systems (e.g., CP/M [Digital]). It must be remembered that this in no way affects the enforcement of the non-discretionary security by the Kernel.

2. Process

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

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

3. Segmentation

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

4. Multiprogramming

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

5. Protection Domains

One of the key elements necessary for valid Kernel implementation is the isolation of the Kernel from all possible outside influences. This can be done through the use of protection domains.

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

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

D. SYSTEM REQUIREMENTS

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

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

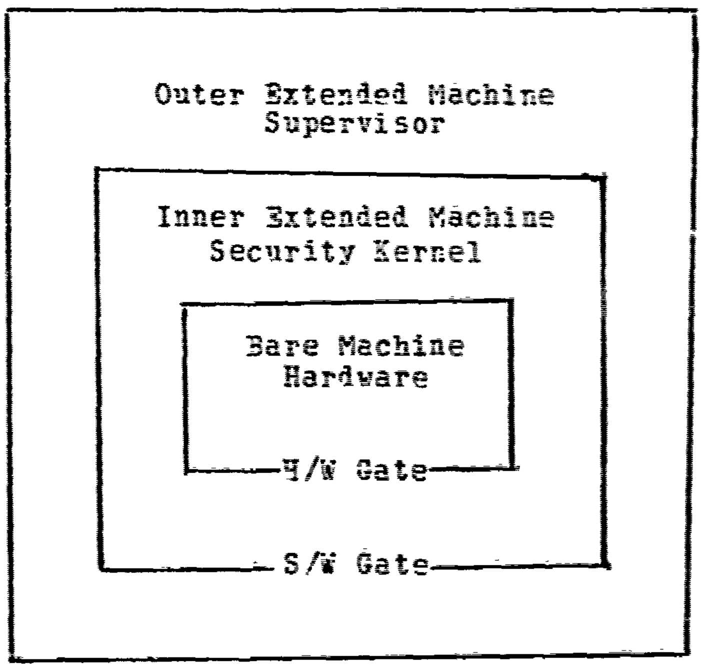


Figure 2. Protection Domains

Only the Kernel has access to privileged machine instructions and controls all system input/output. It provides a segmented environment in which the Supervisor operates. The Supervisor in turn, provides a virtual file environment for the Host computer systems.

II. DESIGN

A. HARDWARE SELECTION

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

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

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

The introduction of the Zilog Z8000 series microprocessor meets both the segmentation and multiple domain requirements. The FSS is targeted for implementation on the Z8001 segmented microprocessor [Zilog(2)] with its associated Memory Management Unit (MMU) [Zilog(1)]. The Z8001 is a 16 bit two-domain machine which produces a 23 bit logical address. The Z8010 MMU maps the 23 bit logical address into a 24 bit absolute address and allows the capability of addressing up to 128 segments (with two MMU's) of 64K bytes each (8M-bytes total) in a two-dimensional memory space. (See [Coleman] for further details.) RS-232 bus compatibility is assumed for serial data input/output at the hardware level. This allows byte synchronization and byte parity checks to be performed at the hardware level by the FSS universal asynchronous receiver-transmitter (UART).

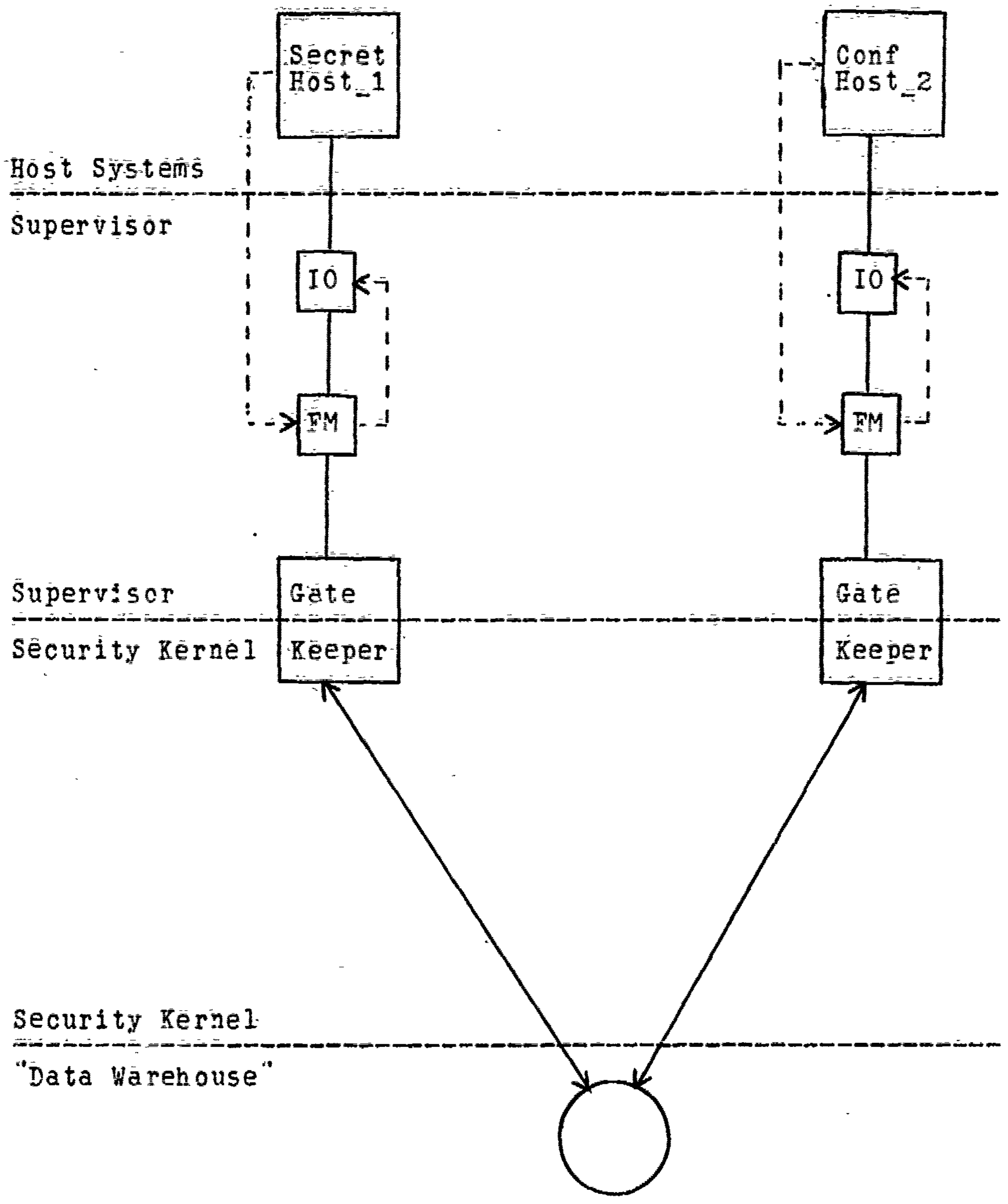
B. SYSTEM STRUCTURE

1. System Levels

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

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

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



------ data
- - - control (i.e., communication)

Figure 3. Abstract System View

Supervisor operates.

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

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

2. System Protocol

Protocols are formal specifications which constrain data exchange between systems and the FSS. These specifications allow the FSS to achieve bounded, deadlock free and fault tolerant communication. To organize and simplify protocol design in the FSS, protocol is logically divided into a hierarchical structure of two interacting layers. Level 1 protocol handles packet (described later) synchronization, error detection, and command type determination. Level 2 handles the repetitive activity of data transfer.

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

3. Host Environment

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

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

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

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

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

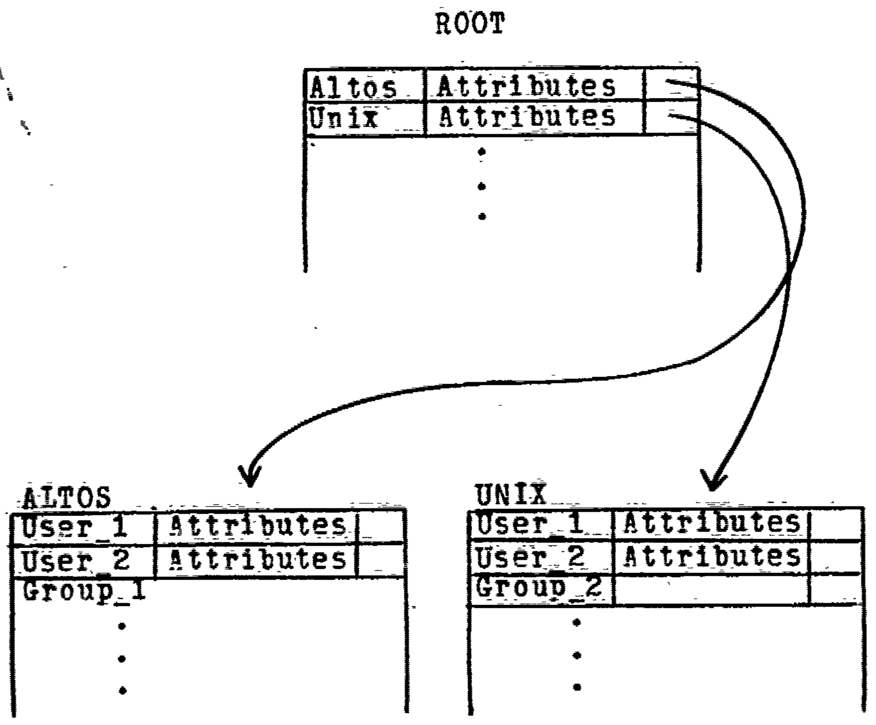


Figure 4. General Supervisor File Hierarchy Example

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

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

Each Host system virtual file hierarchy is constructed from data files and directory files which, as mentioned above, are constructed of Supervisor segments. Although dynamic growth and shrinkage are usual segment attributes, a design choice for System simplification was made to fix segment size at three increments, SMALL (512 bytes), MEDIUM (2K bytes), and LARGE (8K bytes). These sizes were chosen as a compromise between expected file sizes, Supervisor buffer requirements, and minimizing the number of software ring crossings that would be required during a data file "read" or "store" operation. Because segment size is limited and there exists the likelihood of encountering files larger than the maximum segment size. the concept of a multiple segment file (msf) is known to the Supervisor.

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

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

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

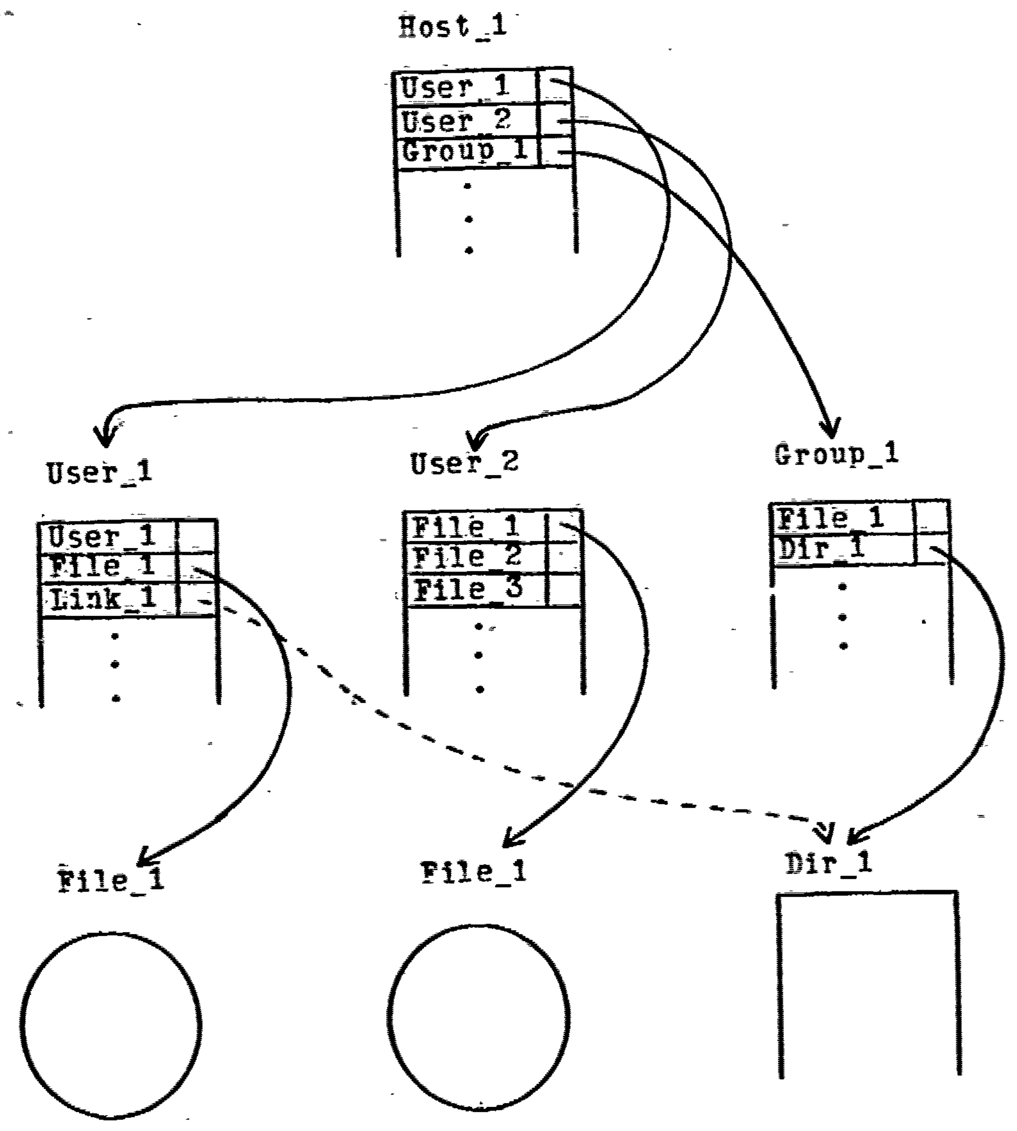


Figure 5. Virtual File Hierarchy (logical view)

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

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

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

a. Directory File

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

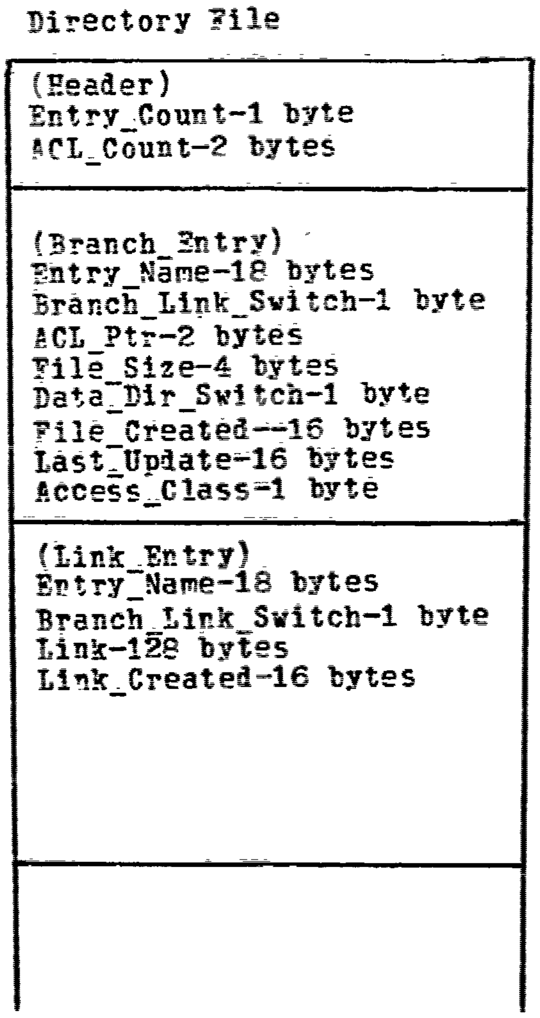


Figure 6. Logical Directory Structure

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

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

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

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

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

Access_Class: This element contains the file access level.

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

ACL_Ptr: This element will point to an ACL for the branch entry. The FSS has only three distinct discretionary access modes: 1) "null" access as the name implies, declares that no access is to be allowed to the specified Host.user combination. 2) "read" access allows a qualified Host.user to read a file only (i.e., no write access). 3) "write" access allows a Host.user write access to a file (also implicit read access). The actual ACL will be a list of authorized users in the form Host.user with an associated access mode. A 'don't care' authorization (in this case a *), will allow general access in that category. For example, *.user would allow the specified 'user' to access this file from any connected Host system with a specified access mode. This ACL for entry "user" can easily be expanded to include other categories such as "project" to further refine the discretionary access allowed to a file.

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

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

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

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

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

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

b. Data Files

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

c. Multiple Segment File Directory

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

4. Host System Commands

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

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

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

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

1) The Supervisor locates the root of the virtual

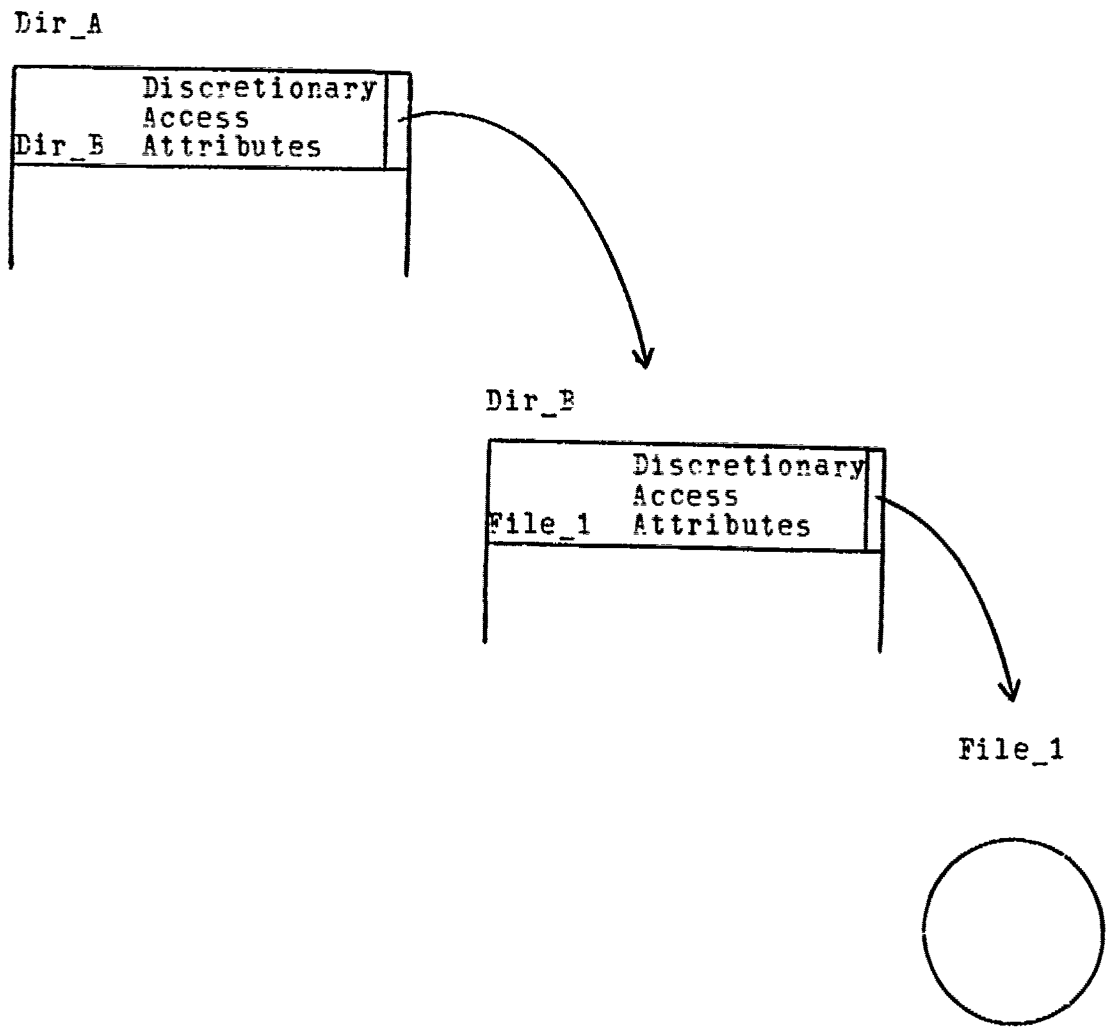

Dir_A
Discretionary Access Attributes
Dir_B
Dir_B
Discretionary Access Attributes
File_1
File_1

Figure 7. File Discretionary Access Control

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

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

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

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

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

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

allowed, an appropriate error code is returned.

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

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

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

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

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

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

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

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

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

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

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

(for directory file)

1) Same.

2) Same.

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

(for file size)

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

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

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

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

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

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

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

READ_ACL <pathname ,userid>. This command is used by the Host systems in conjunction with the ADD_ACL_ENTRY and DELETE_ACL_ENTRY to adjust (give/rescind) the access mode (read/write) allowed to a Host/Host user to a specific file. Actions taken:

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

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

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

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

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

ABORT. This command requests the Supervisor to quit execution of the present command and return the file system to its original state. There are only certain locations in the execution of Host commands that the Supervisor is able to interupt. If an ABORT command is received after an operation has been completed but before the final Host acknowledgement is sent, the original command completion will be acknowledged and the abort command will be ignored. Otherwise, action of the command will be halted and the Supervisor will wait for another Host command. All Host commands (including ABORT) will be explicitly acknowledged with either a "command complete" message or an appropriate error code.

C. PROCESS STRUCTURE

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

Both the FM and IO processes call on Kernel primitives to perform actual segment manipulation. The normal order in which these calls are made is fixed by the Kernel design. To add a segment to a process memory, the order of Kernel calls is: 1) Gatekeeper.Create_Segment, 2) Gatekeeper.Make_Known, and 3) Gatekeeper.Swap_In. To delete a segment from a process memory, the order of Kernel calls is: 1) Gatekeeper.Swap_Out, 2) Gatekeeper.Terminate, and 3) Gatekeeper.Delete_Segment. The Supervisor procedures use these invokation orders.

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

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

The modular construction of the FSS enhances System structure. All data bases, except the files themselves, are module local. Code is expected to be written in PLZ/SYS [Snook], which is a high level pascal-like structured programming language. Because of the its length, code is located in Appendix C. The code listed in this appendix gives the interprocess and intermodule control structure of the FSS.

1. Shared Segment Interactions

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

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

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

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

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

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

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

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

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

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


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

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

These eventcounts and sequencers, with their associated Kernel primitives, are used by the FSS to perform the synchronization functions of Block and Wakeup [Coleman], described in the original Kernel design. Eventcounts and sequencers provide a clearer picture of the process interaction as well as explicit control of the 'readers/writers' problem. Even more importantly, they permit the synchronization between processes of different access levels. This is essential in order to permit a high level Host to read files of a lower level.

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

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

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

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

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

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

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

2. File Management Process

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

a. File Management Command Handler Module

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

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

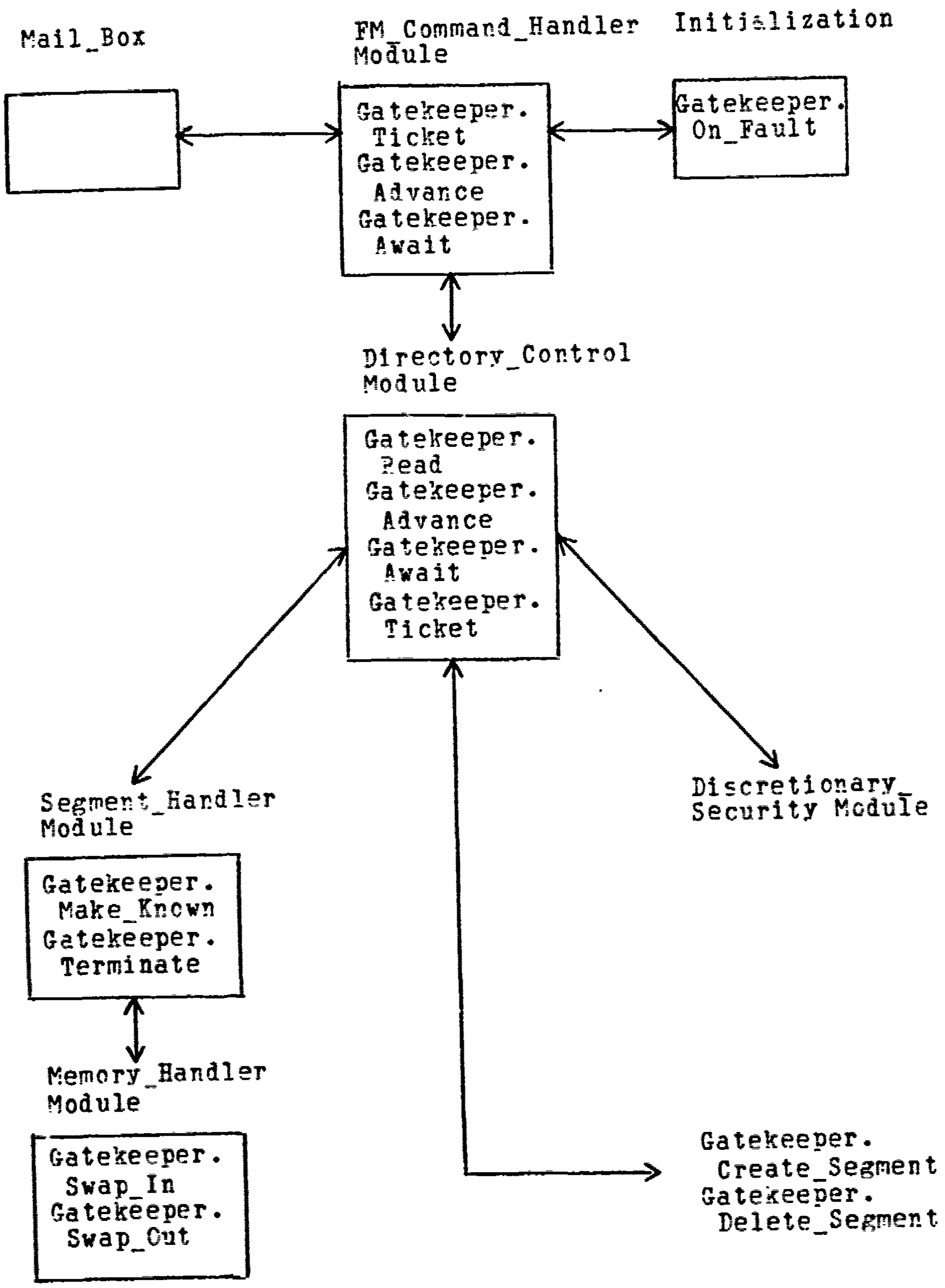
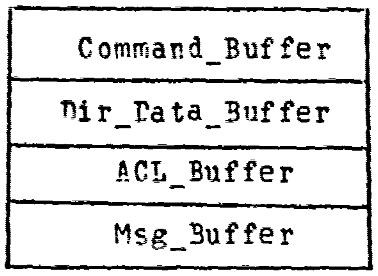


Figure 8. FM Process Modules



Mail_Box Segment

```
Mail_Box Record [

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

Figure 9. Mail_Box Segment

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

Figure 10. Command_Handler Module Procedure Input/Output parameters

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

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

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

pathname

link

file type

command type

file size

access level

userid

ACL entry.

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

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

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

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

File size is passed by the Host during STORE_FILE requests. This information is necessary for the FM process to create a temporary file of sufficient size to store a Host file. File size is relayed to the IO process so that the IO process can go directly to the data file without having to check the directory file for file size. File size is in bits.

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

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

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

b. Directory Control Module

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

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

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

Figure 11. Directory_Control Module Procedures Input/Output Parameters

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

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

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

DELETE_FILE

CREATE_FILE

CREATE_LINK

READ_FILE (dir, size)

READ_ACL

ADD_ACL_ENTRY

DELETE_ACL_ENTRY.

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

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

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

If no error condition occurs during the

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

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

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

Once this check has been satisfactorily completed, and room exists in the directory, the Kernel call Gatekeeper.Create_Segment is made to create the file. The initial file size is zero for data files since the Supervisor has no prior knowledge of the size of the file that will be stored in the branch entry. As explained earlier, a file size of LARGE (8K bytes) was selected for the fixed directory size.

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

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

The last three Host requests handled by the Dir_Cntrl_Directory procedure are related. Again, appropriate discretionary access checks must be made in the parent directory. If no error condition arises, the action taken is straight foreward. In the case of the READ_ACL command, the file ACL is transfered to the mail_box ACL_buffer and the procdure returns to the FM_Command_Handler module. In the case of the ADD(DELETE)_ACL_ENTRY commands, the action is completed by the Dir_Cntrl_Directory procedure and the appropriate Dir_Succ_Code returned.

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

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

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

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

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

c. Discretionary Security Module

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

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

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

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

Figure 12. Discretionary_Security Module Procedure Input/Output Parameters

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

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

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

d. Segment Handler Module

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

Pathname	Seg_#	Mode	Use

Figure 13. FM_KST

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

Figure 14. Segment_Handler Module Procedure Input/Output Parameters

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

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

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

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

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

e. Memory Handler Module

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

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

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

Figure 15. Memory_Handler Module
Procedure Input/Output Parameters

Seg_#	Size	Base_Addr	Use

Figure 16. FM_AST

0	1	2	. . .	Base_Addr
				SEG_#

Figure 17. Mem_Map

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

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

A preliminary analysis of memory requirements indicates that process linear virtual memory will need to be at least 24K bytes. The driving factor in this calculation is the fact that two data segments (possibly 8K bytes each) may be required in process memory during the copying of a data file into the data file "buffer". A 24K byte memory would allow for the worst case, viz., one 8K byte segment positioned in the middle of linear memory; room would still exist for the two 8K byte segments.

3. Input/Output Process

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

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

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

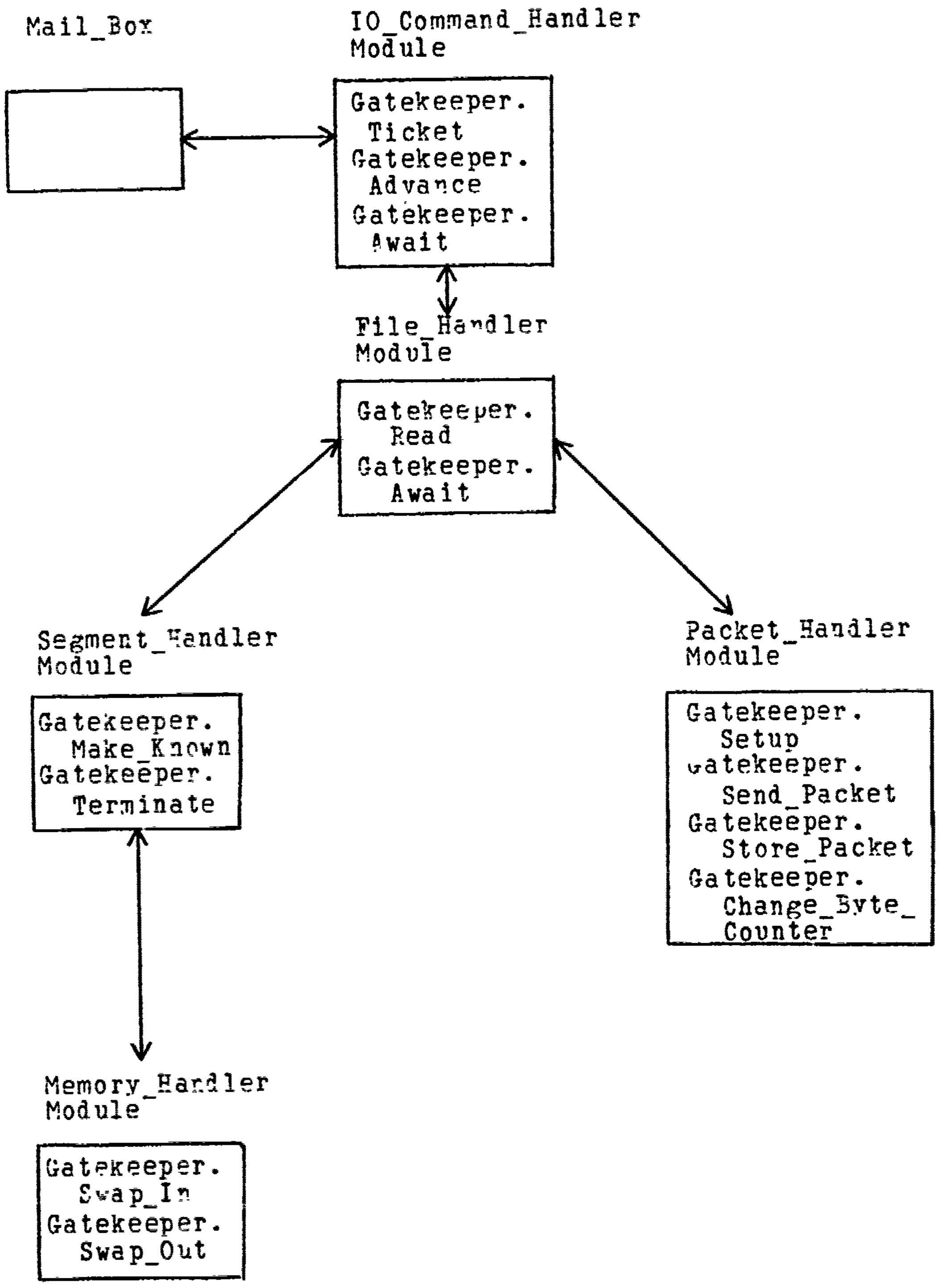
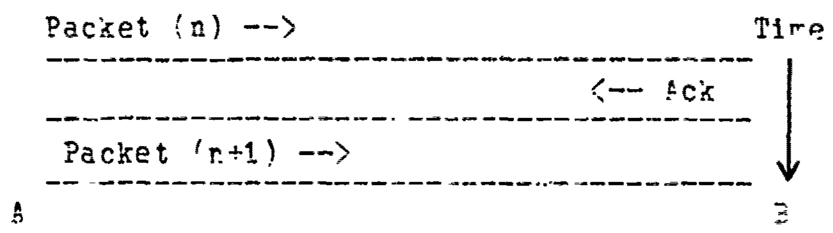


Figure 18. IO Process Module

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

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

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



DATA PACKET

Packet_Type	Byte
Packet_Number	Lword
Data	512 Bytes
Check_Sum	Lword

COMMAND PACKET

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

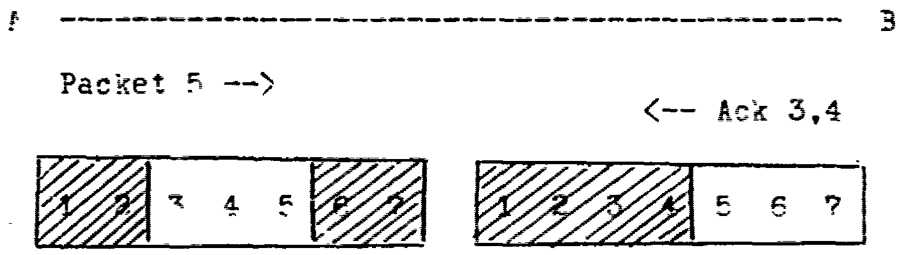
Figure 19. Packet Construction

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

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

For an explanation of how the packet window works, let N(t) denote the transmitted sequence number of the current packet and let N(t+1) denote the next expected sequence number. The window width is denoted by W. At the start of communication, e.g., when a Host sends a command to the FSS, the Host is allowed to transmit packets bearing sequence numbers in the range $0 \leq N(t) < W$. The receiver expects the packets to arrive in correct sequential order. As they arrive, packets are checked for correctness (at both the hardware (USART) and software level); an incorrect packet is discarded and may be considered 'lost'. Let the sequence number of a particular correctly received packet be S. If $S=N(t+1)$ (i.e., the expected packet), then the packet is received in the correct sequence and it should be accepted by the receiver and an acknowledgement sent with the proper sequence number (in this case, S) to the sender. If $S<N(t+1)$, then the packet is a repetition of a packet previously received by the receiver; the second transmission may be due to either a lost or delayed acknowledgement. The receiver should generate another acknowledgement and send it to the sender and otherwise ignore the packet. If $S>N(t+1)$, then the packet is ahead of sequence, indicating that an earily pack. has been lost; such a packet should be ignored and an "error" acknowledgement sent so the packet can be retransmitted.

The arrival of acknowledgements at the sender also needs to be discussed. As each acknowledgement arrives, the sender can delete the copy it has retained of the corresponding packet. As packets are acknowledged, fresh packets can be transmitted, i.e., when packet 0 has been acknowledged, packet W can be sent. Acknowledgements can get lost in transmission as well as packets. If a received acknowledgement does not refer to the earliest transmitted packet awaiting acknowledgement, then, in this protocol, the sender may safely delete all packets up to and including that referenced by the acknowledgement. Against each copy of a transmitted packet will be noted a time (i.e., the time-out) by which time the packet must be acknowledged. Failing such an acknowledgement, the packet must be retransmitted with its original sequence number. A packet will only be received in sequential order, so it will be necessary to retransmit not only the earliest unacknowledged packet, but also all later packets. The following figure illustrates this protocol. The queues should be considered as circular with automatic wrap-around.



In this figure, the sender is node A and the receiver is node B. Node A has sent out packets 3,4, and 5, the last of which is still in transit to B. Node B has received all packets up to and including 4. It has just acknowledged 3 and 4 and is ready to accept 5,6, and 7 when they arrive in order. When node A receives acknowledgement for 3 and 4, it will be able to transmit successfully packets 6 and 7.

This protocol insures that packets are handled in sequential order which will insure that the data is received and stored correctly. It also assures positive control over the receipt and transmission of packets; a necessary requirement to prevent buffer overflow and loss of data.

The Kernel controls all the hardware assets, as explained in Chapter 1. Kernel calls are therefore necessary to transfer packets between the FSS and the Host systems. The format of these Kernel calls are:

Gatekeeper.Setup (Buff_Addr, Mode, Status)
Gatekeeper.Send_Packet (Offset, Status)
Gatekeeper.Store_Packet (Offset, Status)
Gatekeeper.Change_Byte_Counter (#_of_Bytes, Status)

Each hardware port is virtualized into an input and an output port. Each virtual port has associated with it a unit control block (UCB) at the Kernel level. The elements of these UCB's are:

Byte_Counter: This element is used to keep track of the number of bytes that have been transmitted or received. This counter is modulo "packet size" so that once packets are synchronized, they should remain so. It can be altered by the Change_Byte_Counter call in order to get the FSS and Host back into packet synchronization.

Buffer_Address: This is the starting address in the input/out buffer where packets will be placed (incomming) or taken from (outgoing). It is initialized by the Setup Kernel call.

Buffer_Length: This element is the length (in packets) of the input/output buffer. This allows the Kernel to perform automatic wrap around at the end of the buffer.

Window_Width: This element is used by the input port UCB to prevent buffer over flow. Each invocation of Store_Packet will advance the window and allow another packet to be stored into the IO buffer. If a Host system violates protocol by sending too many packets, the Kernel will dump them to a "bit bucket". This element is used by the output port to control the number of packets that the FSS is able to send to a Host before receiving an acknowledgement. Although this parameter (viz., window width) may be different for the various Host systems, it should not change often and can therefore be set at system initialization.

For a store operation (FSS to receive packets), a Setup call is used to set the input UCB base address to the initial storage location in the IO buffer. A Setup call is also required to set the output UCB with the base address in the IO buffer from which acknowledgments will be sent. It should be noted here that the IO buffer in the IO process is the location that packets are checked for errors and "enpacketed" or "depacketed". It is just a intermdiate stop for data and neither the final destination nor origin of data.

Subsequent Kernel calls to Store_Packet will return the location of the next packet in the IO buffer to be processed. The Kernel will store ahead into the IO buffer during the store operation but will not over write the buffer. That is, each call to the Kernel will indicate that a new packet location is open. The IO process will control which packets (and how many) are sent to the FSS by proper use of acknowledgements (for both correct and incorrect packets).

Two Setup calls are also necessary for a send operation. They again set the virtual input/output ports for the transfer of packets from the FSS to a Host. Subsequent calls to Send_Packet indicate that a Packet is ready to be transmitted. The IO process knows when it can discard a packet by the acknowledgments it receives from the Host system.

The Change_Byte_Counter primitive is used by the synchronization procedure to shift a UCB byte counter in order to bring packet transmission back into synchronization. (Synchronization may be required during a temporary communication interruption or system start up.)

The following is a description of the three "new" modules which make up the IO process.

a. Input/Output Command Handler Module

At the top of the IO process module hierarchy is the IO_Command_Handler module (see Appendix C, p. 117). This module is responsible for the interface with the PM process. Communication between the processes is via the mail_box shared segment and synchronization is through the use of an eventcount and the Kernel primitives TICKET, ADVANCE and AWAIT. The procedures of this module along with their input/output parameters are listed in figure 20.

The IO_Cmd_Hnd procedure, like the FM_Cmd_Hnd procedure a case statement, routes FM process instructions to a specific IO_Command_Handler procedure for action.

The procedure involved when the Host command is not a READ_FILE or STORE_FILE request is the IO_Cmd_Hnd_Ack procedure. This procedure is able to invoke the Packet_Handler module directly for performing directed (by the FM process) Host acknowledgement and/or data transfer from the shared mail_box segment.

The IO_Cmd_Hnd_Send and IO_Cmd_Hnd_Store procedures are relatively straight forward. They provide the IO-FM process interface required for a READ_FILE or STORE_FILE Host request. Both procedures call the File_Handler module to perform the actual file manipulation.

b. File Handler Module

The File_Handler module is required for file manipulation in the IO process and is the level in the IO process at which files are known. The procedures which make up this module along with their input/output parameters are listed in figure 21. As mentioned above, there are only two Host requests that require the IO process to bring data files into process memory. These are READ_FILE and

PROCEDURE	INPUT	OUTPUT
IO_Cmd_Hnd	Mail_Box.Msg.Inst	Output returned
IO_Cmd_ Hnd_Ack	Mail_Box.Msg.Succ_Code	from subordinate modules.
IO_Cmd_ Hnd_Send	Mail_Box.Msg.Pathname Mail_Box.Msg.File_Size	
IO_Cmd_ Hnd_Store	Mail_Box.Msg.Pathname Mail_Box.Msg.File_Size	

Figure 20. IO_Command_Handler Module
Procedure Input/Output parameters

PROCEDURE	INPUT	OUTPUT
File_Hnd_ Send_File	Mail_Box.Msg.Pathname Mail_Box.Msg.File_Size	File_Succ_Code
File_Hnd_ Store_File	Mail_Box.Msg.Pathname Mail_Box.Msg.File_Size	File_Succ_Code

Figure 21. File_Handler Module
Procedure Input/Output Parameters

STORE_FILE. Note that since file size is passed from the FM process, and the the access to the data files involved is controlled in the FM process, data file segments can be brought directly into IO process memory and any requirement for the IO process to access directory files (other than tree traversal) is eliminated. Because the terminal nodes in the tree traversal are controlled by the FM process, the paths to these terminal nodes will not be alterable until control is released by the FM process.

The File_Handler module consist of two procedures, File_Hnd_Send_File (for Host command READ_FILE) and File_Hnd_Store_File (for Host command STORE_FILE). Both procedures operate in a similar manner. Upon receiving a pathname and file size from the FM process, these procedures use the Segment_Handler procedures to bring the necessary data file (segment(s)) into process memory. A call is then made to the Packet_Handler module to transfer data from/to specified segments.

The order of events in the reading and storing of data files follows the following sequences. For a READ_FILE operation, the order of actions taken by the Supervisor are:

1) Discretionary and non-discretionary checks are made in the FM process.

2) A copy is made of the data file into a per-process data file buffer.

3) The pathname of the data file to be read

(remember, directory data is read by the FM process) is passed to the IO process along with the file size. The IO process can determine the file size from the file directory but by passing file size to the IO process, this step is eliminated for the IO process.

4) The read takes place in the IO process.

5) The IO process returns to the FM process with a success code of "read complete" or an appropriate error code. The only reason for a read operation to fail in the IO process is the receipt of an abort command from the Host or a "time out" which would occur if the Host stopped sending for some unexplained reason.

6) The FM process instructs the IO process to acknowledge the "read complete" or to send the appropriate error code. The data file read buffer is then free for further use.

If the operation is a STORE_FILE operation the following steps are taken by the Supervisor:

1) Discretionary and non-discretionary security checks are made by the FM process.

2) A temporary file is created by the Supervisor large enough to store the file in. Appropriate use of the synchronization primitives prevents this temporary file from being used by more than one process at a time.

3) The pathname of the temporary file is sent to the IO process and the IO process stores the file into the temporary file.

4) The IO process returns a success code to the FM process and the FM process updates the directory to reflect the new file (viz., Entry_Name of temporary file is changed to the old file Entry_Name). The old file is then deleted by the FM process.

5) The FM process then instructs the IO process to acknowledge the "store complete". There is no reason a store operation should fail other than an explicit abort request by the Host system or hardware failure.

c. Packet Handler Module

The Packet_Handler module does the actual transfer of data between the FSS and the Host system and is the IO process level at which the concept of "packet" is known. The procedures of this module along with their input/output parameters are listed in figure 22. The tasks that this module must perform are: 1) synchronization of packets, 2) error detection, 3) packet acknowledgement, and 4) transfer of data to/from Supervisor segments. Figure 23 is a finite state diagram of packet transfer.

The synchronization task is performed on the system IPL and whenever packet synchronization is lost thereafter. Error detection and request for retransmission upon error detection are complimentory functions which are performed on every packet received from a Host.

Packet transfer during synchronization procedures is in groups of three. This allows the

PROCEDURE	INPUT	OUTPUT
Pk_Hnd_Sync	Sync Cmd	Packet Sync
Pk_Hnd_ Ack	Packet Mail_Box.Msg.Succ_Code	Pk_Succ_Code
Pk_Hnd_ Send	Data	Packet
Pk_Hnd_ Store	Packet	Data

Figure 22. Packet_Handler Module Procedure Input/Output Parameters

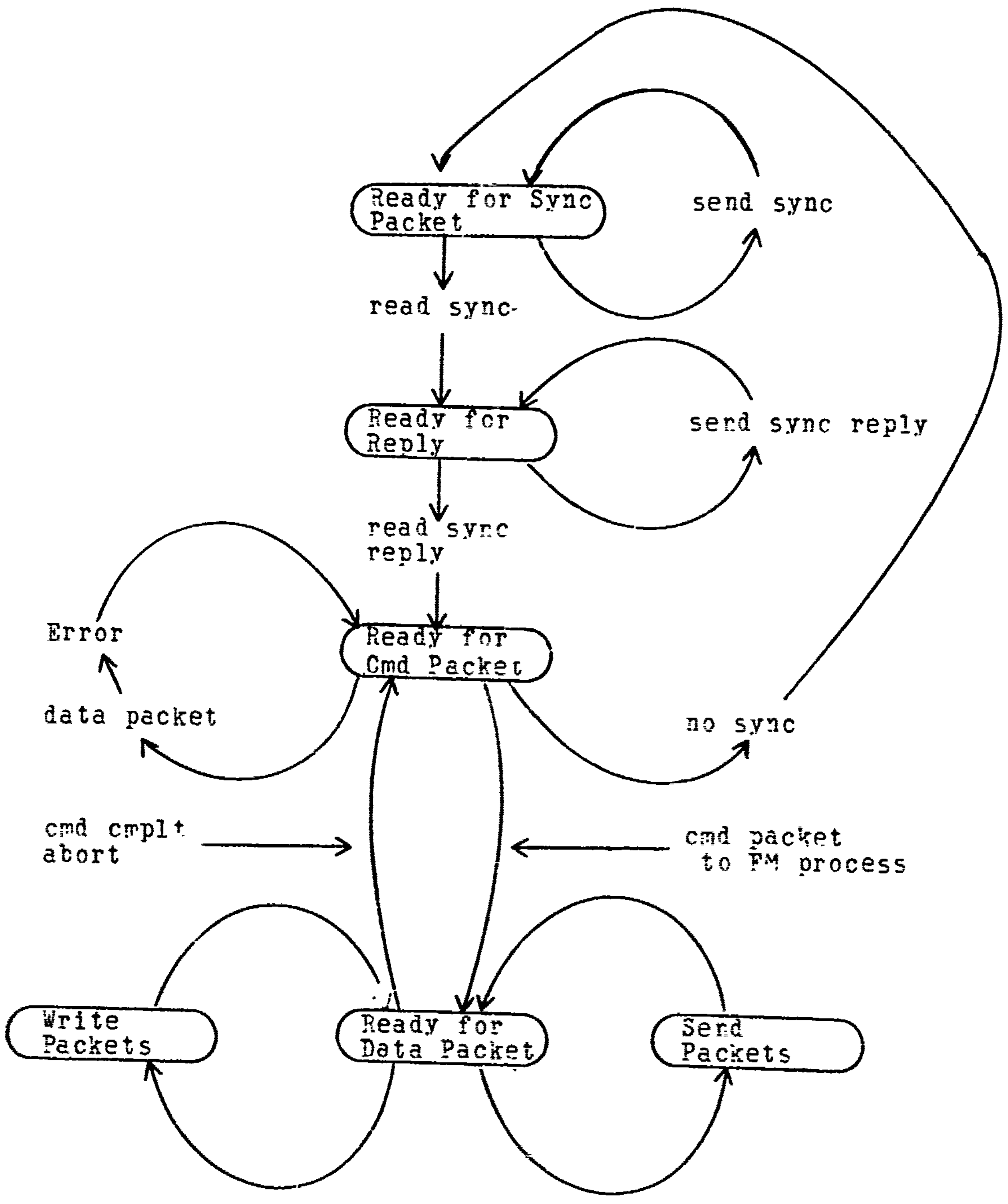


Figure 23. Finite State Diagram of Packet Transfer

synchronization procedure to begin synchronization in the middle of the first packet and still have two packets to confirm synchronization when it is achieved.

Packet transfer of command packets occurs one at a time. The reason for this is that each command packet must be acted upon in a synchronous manner. Data packet read ahead and write behind is permitted to increase the transfer rate of data packets. The number of packets that are allowed to be sent or stored depends on the IC buffer size. The Packet_Handler module is also responsible for data "enpacketing" and "depacketing" for the FSS.

The Pk_Hnd_Sync procedure is used to synchronize packet transmission. It is explicitly called at IPL and whenever the packet synchronization is lost by the Host system. It is invoked implicitly by the FSS whenever a packet is not able to be decoded (viz., the packet type and packet check-sum are incorrect).

The Pk_Hnd_Ack procedure is used to send acknowledgements to the Host systems. This procedure will always be called from the IO_Command_Handler module which will require the Packet_Handler module to either acknowledge the Host with a specific message or to send some data located in a mail_box segment buffer to the Host.

The Pk_Hnd_Send procedure is used to transfer data segments from the FSS to a Host system. This procedure is called from the File_Handler module which makes sure that the correct data segment is in process memory for the transfer. The segment number along with the number of bits that are reouired to be transfered are passed to this procedure from the File_Handler module. This procedure then transfers the segment until the specified number of bits have been transmitted. A success code is returned when action is complete.

The Pk_Hnd_Store procedure works in a manner completely analogous to the Pk_Hnd_Read procedure.

III. CONCLUSIONS

A. STATUS OF RESEARCH

This design applies state of the art software and hardware to solve the secure multilevel computer problem in a file storage system. It presents an inexpensive but highly powerful design for a system based on a micro-computer but not restricted to a micro-computer environment, i.e., there is no restictior on the type of Host computer system serviced by the FSS. Implementation of this design on Z8000 hardware along with the analysis of FSS design parameters (Appendix A) are tasks left to be done.

There are two major classes of applications for the FSS. One application uses the FSS as a system file system (e.g., for distributed micros). This implies that the total system is multilevel secure with only one secure component (viz., the Kernel). It must be noted, however, that in this configuration, the distributed Hosts (i.e., the micros) have no autonomous life.

The other class of applications. involves using the FSS as one element of a net of autonomous Host systems. In this configuration, the FSS provides facilities for controlled data sharing and communication.

An obvious direct application of the FSS, is for shipboard use (e.g., for the SNAP-II system [Smith]) or for use at other installations where data would be more efficiently used if controlled data sharing were allowed.

A major design choice of the FSS which allowed the Kernel to be kept small (and therefore more easily verifiable), was the elimination of the discretionary security from the Kernel domain to the Supervisor domain. The implication of this choice is that each Host system is responsible for its own discretionary security; not an unreasonable request or design choice.

The next major task to be accomplished in this project is FSS implementation. This will not be a trival task, but it is felt that the designs presented in this thesis and the companion work done by Coleman provide a solid basis.

B. FOLLOW ON WORK

This design is a specific implementation of one member of a family of operating systems based on the Security Kernel concept discussed by O'Connell and Richardson [O'Connell]. There are obvious areas that this design could be expanded and generalized; areas that should be examined after a successful first implementation. Some of these areas are:

'operator' terminal interface funcions

expanded Host commands

map of different 'user' names in different Hosts to a common "user" in the FSS

data compaction onto secondary storage

multilevel Hosts

moving discretionary security into the Kernel domain dynamic process creation/deletion.

These are just a few of the many possible areas for expansion that could be explored. One area not mentioned in the list but an area that should be looked at during the initial implementation is for a way to prevent the Supervisor from suffering a 'segment fault'. The present arrangement, with a fault handler, is not efficient or 'elegant'. Since the deletion of a segment is controlled by the Delete_Segment Kernel primitive, a method of leaving an 'orphan' copy in process memory would eliminate the fault condition. The only operation that would be defined on this 'orphan' would be a Delete_Segment command by a process to remove it from process memory. After it had been deleted by all processes, the copy could be destroyed. A variation of this scheme would, upon a Kernel Swap_In call, swap into process memory a per-process copy of the desired segment. Swap_Out would be used to free process memory.

APPENDIX A--SYSTEM PARAMETERS

SMALL	Segment size	512 bytes
MEDIUM	Segment size	2K bytes
LARGE	Segment size	8K bytes
MAX_FILE_SIZE	Max file size	256K bytes
MAX_ENTRY	Max dir entries	32
ACL_POOL	Max Acl_Entries per directory	1024
Pathlength	Max	128 bytes
Entry_Name	Size	18 bytes
Known Segments	Max per process	--

APPENDIX B--SUCCESS AND ERROR CODES

CODE	LOCATION
File Deleted	FM_Command_Handler
File Created	Module
Link_Created	
Store_Complete	
Read_Complete	
ACL_Read_Complete	
ACL_Entry_Added	
ACL_Entry_Deleted	
Cmd_Aborted	
Cmd_Packet_Expected	
Illegal_Cmd	
Illegal_Cmd_Format	
File_Not_Found	Directory_Control
Not_Terminal_File	Module
Write_Access_Not_Allowed	Discretionary_Security
Read_Access_Not_Allowed	Module
Time_Out	Packet_Handler
No_Sync	Module
Packet_Ack	
Packet_Error	

APPENDIX C

```
FM_COMMAND_HANDLER MODULE

CONSTANT
  FALSE := 0
  TRUE := 1
  NULL := 2

EXTERNAL

DIR_CNTRL_DIRECTORY PROCEDURE (MSG           BYTE
                                USERID        BYTE
                                PATHNAME      STRING
                                FILE_TYPE     BYTE
                                ACCESS_LEVEL  BYTE
                                LINK          STRING
                                ACL_ENTRY     ACL_TYPE)
  RETURNS (DIR_SUCC_CODE                      BYTE)
!for host cmds that require parent directory:
 delete_file,
 create_file,
 create_link,
 read_acl,
 add_acl_entry,
 delete_acl_entry!

DIR_CNTRL_DATA PROCEDURE (MSG          BYTE
                          USERID       BYTE
                          PATHNAME     STRING)
                          FILE_SIZE    LWORD)
  RETURNS (DIR_SUCC_CODE               BYTE
           DIR_PATHNAME                STRING)
!for host cmds that access data file:
 read_file,
 store file!

DIR_CNTRL_UPDATE PROCEDURE (MSG          BYTE
                            USERID       BYTE
                            PATHNAME     STRING)
  RETURNS (DIR_SUCC_CODE                 BYTE)
!to update directory after io process
 acts on host cmds: read_file, store_file,
 abort!
```

```
GLOBAL !module entry point!

FM_CMD_HND PROCEDURE !case statement of Host cmds!
ENTRY
DO
  MAIL_BOX.MSG.INST := READ_CMD
  MAIL_BOX.MSG.PATHNAME := NULL
  MAIL_BOX.MSG.FILE_SIZE := NULL
  MAIL_BOX.MSG.SUCC_CODE := NULL
  t := GATEKEEPER.TICKET (MAIL_BOX, C)
  GATEKEEPER.ADVANCE (MAIL_BOX, S)
  GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  IF MAIL_BOX.MSG.INST = CMP_PK_READY
    THEN
    IF HOST_CMD
      CASE DELETE_FILE THEN FM_CMD_HND_DELETE_FILE
      CASE CREATE_FILE THEN FM_CMD_HND_CREATE_FILE
      CASE CREATE_LINK THEN FM_CMD_HND_CREATE_LINK
      CASE READ_FILE THEN FM_CMD_HND_READ_FILE
      CASE STORE_FILE THEN FM_CMD_HND_STORE_FILE
      CASE READ_ACL THEN FM_CMD_HND_READ_ACL
      CASE ADD_ACL_ENTRY THEN FM_CMD_HND_ADD_ACL_ENTRY
      CASE DELETE_ACL_ENTRY THEN FM_CMD_HND_DELETE_ACL_ENTRY
      CASE ABORT THEN FM_CMD_HND_ABORT
    ELSE
      MAIL_BOX.MSG.INST := ACK_HOST
      MAIL_BOX.MSG.PATHNAME := NULL
      MAIL_BOX.MSG.FILE_SIZE := NULL
      MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (ILLEGAL_CMD)
      t := GATEKEEPER.TICKET (MAIL_BOX, C)
      GATEKEEPER.ADVANCE (MAIL_BOX, C)
      GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
    FI
  ELSE
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (CMD_PK_EXPECTED)
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  FI
OD
```

```
INTERNAL
MSG = BYTE

FM_CMD_HND_DELETE_FILE PROCEDURE
ENTRY
  MSG := DELETE_FILE
  DIR_CNTRL_DIRECTORY (MSG
                       USERID
                       PATHNAME
                       NULL !file_type!
                       NULL !access_level!
                       NULL !link!
                       NULL) !acl_entry!
  !returns dir_succ_code!
  IF DIR_SUCC_CODE = TRUE
    THEN
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := FILE_DELETED
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  ELSE
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
    !file not found; write access to directory
     not permitted!
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  FI
END FM_CMD_HND_DELETE_FILE
```

```
FM_CMD_HND_CREATE_FILE PROCEDURE
ENTRY
  MSG := CREATE_FILE
  DIR_CNTRL_DIRECTORY (MSG
                       USERID
                       PATHNAME
                       FILE_TYPE
                       ACCESS_LEVEL
                       NULL !link!
                       NULL) !acl_entry!
  !returns dir_succ_code!
  IF DIR_SUCC_CODE = TRUE
    THEN
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := FILE_CREATED
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  ELSE
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
    !directory not found; write access to directory
     not permitted; directory full!
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  FI
END FM_CMD_HND_CREATE_FILE
```

```
FM_CMD_CREATE_LINK PROCEDURE
ENTRY
  MSG := CREATE_LINK
  DIR_CNTRL_DIRECTORY (MSG
                       USERID
                       PATHNAME
                       NULL !file type!
                       NULL !access level!
                       LINK
                       NULL) !acl_entry!
  !returns dir_succ_code!
  IF DIR_SUCC_CODE = TRUE
    THEN
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := LINK_CREATED
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C (t+2))
  ELSE
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
    !directory not found; write access to directory
     not permitted; directory full!
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  FI
END FM_CMD_HND_CREATE_LINK
```

```
FM_CMD_READ_FILE PROCEDURE
ENTRY
  IF FILE_TYPE = DATA
    THEN
    MSG := READ_FILE
    DIR_CNTRL_DATA (MSG
                    USERID
                    PATHNAME
                    NULL) !file_size!
    !returns dir_succ_code, dir_pathname, dir_file_size!
    IF DIR_SUCC_CODE = TRUE
      THEN
      MAIL_BOX.MSG.INST := READ_FILE
      MAIL_BOX.MSG.PATHNAME := DIR_PATHNAME
      MAIL_BOX.MSG.FILE_SIZE := DIR_FILE_SIZE
      MAIL_BOX.MSG.SUCC_CODE := NULL
      t := GATEKEEPER.TICKET (MAIL_BOX, C)
      GATEKEEPER.ADVANCE (MAIL_BOX, C)
      GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
      IF MAIL_BOX.MSG.SUCC_CODE = TRUE
        THEN
        MSG := UPDATE_READ
        DIR_CNTRL_UPDATE (MSG
                          USERID
                          PATHNAME)
        !update will not fail!
        MAIL_BOX.MSG.INST := ACK_HOST
        MAIL_BOX.MSG.PATHNAME := NULL
        MAIL_BOX.MSG.FILE_SIZE := NULL
        MAIL_BOX.MSG.SUCC_CODE := READ_COMPLETE
        t := GATEKEEPER.TICKET (MAIL_BOX, C)
        GATEKEEPER.ADVANCE (MAIL_BOX, C)
        GATEKEEPER.AWAIT (MAIL_BOX, C)
      ELSE
        MAIL_BOX.MSG.INST := ACK_HOST
        MAIL_BOX.MSG.PATHNAME := NULL
        MAIL_BOX.MSG.FILE_SIZE := NULL
        MAIL_BOX.MSG.SUCC_CODE := MAIL_BOX.MSG.SUCC_CODE
        !error code returned from io process!
        !file not found by io process;
         file read aborted by  write;
         file read aborted by file deletion;
         cmd packet received!
        t := GATEKEEPER.TICKET (MAIL_BOX, C)
        GATEKEEPER.ADVANCE (MAIL_BOX, C)
        GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
      FI
    ELSE
      MAIL_BOX.MSG.INST := ACK_HOST
      MAIL_BOX.MSG.PATHNAME := NULL
      MAIL_BOX.MSG.FILE_SIZE := NULL
      MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
      !file not found;
       read access to file not permitted!
      t := GATEKEEPER.TICKET (MAIL_BOX, C)
      GATEKEEPER.ADVANCE (MAIL_BOX, C)
      GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
    FI
  ELSE
    IF FILE_TYPE = DIRECTORY
      THEN
      MSG := READ_DIR
      DIR_CNTRL_DIRECTORY (MSG
                             USERID
                             PATHNAME)
                             NULL !file_type!
                             NULL !access_level!
                             NULL !link!
                             NULL) !acl_entry!
       !returns dir_succ_code!
       IF DIR_SUCC_CODE = TRUE
         THEN
         MAIL_BOX.MSG.INST := ACK_HOST
         MAIL_BOX.MSG.PATHNAME := NULL
         MAIL_BOX.MSG.FILE_SIZE := NULL
         MAIL_BOX.MSG.SUCC_CODE := DIR_READ_COMPLETE
         !dir data transfered from dir_buffer;
          acknowledgement sent!
         t := GATEKEEPER.TICKET (MAIL_BOX, C)
         GATEKEEPER.ADVANCE (MAIL_BOX, C)
         GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
        ELSE
         MAIL_BOX.MSG.INST := ACK_HOST
         MAIL_BOX.MSG.PATHNAME := NULL
         MAIL_BOX.MSG.FILE_SIZE := NULL
         MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
         !directory not found,
          read access to directory not permitted!
         t := GATEKEEPER.TICKET (MAIL_BOX, C)
         GATEKEEPER.ADVANCE (MAIL_BOX, C)
         GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
        FI
      ELSE
        MSG := READ_ENTRY_DATA
        DIR_CNTRL_DIRECTORY (MSG
                               USERID
                               PATHNAME
                               NULL !file_type!
                               NULL !access_level!
                               NULL !link!
                               NULL) !acl_entry!
        !returns dir_succ_code!
        IF DIR_SUCC_CODE = TRUE
          THEN
          MAIL_BOX.MSG.INST := ACK_HOST
          MAIL_BOX.MSG.PATHNAME := NULL
          MAIL_BOX.MSG.FILE_SIZE := NULL
          MAIL_BOX.MSG.SUCC_CODE := ENTRY_READ_COMPLETE
          !entry data transfered from dir_buffer;
           acknowledgement sent!
          t := GATEKEEPER.TICKET (MAIL_BOX, C)
          GATEKEEPER.ADVANCE (MAIL_BOX, C)
          GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
        ELSE
          MAIL_BOX.MSG.INST := ACK_HOST
          MAIL_BOX.MSG.PATHNAME := NULL
          MAIL_BOX.MSG.FILE_SIZE := NULL
          MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
          !file not found; read access to file not permitted!
          t := GATEKEEPER.TICKET (MAIL_BOX, C)
          GATEKEEPER.ADVANCE (MAIL_BOX, C)
          GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
        FI
      FI
    FI
  END FM_CMD_HND_READ_FILE
```

```
FM_CMD_HND_STORE_FILE PROCEDURE
ENTRY
  MSG := STORE_FILE
  DIR_CNTRL_DATA  (MSG
                   USERID
                   PATHNAME
                   FILE_SIZE)
  !returns dir_pathname; dir_succ_code!
  IF DIR_SUCC_CODE = TRUE
    THEN
    MAIL_BOX.MSG.INST := STORE FILE
    MAIL_BOX.MSG.PATHNAME :=   DIR_PATHNAME
    MAIL_BOX.MSG.FILE_SIZE :=   FILE_SIZE
    MAIL_BOX.MSG.SUCC_CODE    NULL
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
    IF MAIL_BOX.MSG.SUCC_CODE = TRUE
      THEN
      MSG := UPDATE_STORE
      DIR_CNTRL_UPDATE (MSG
                        USERID
                        PATHNAME)
      !update will not fail!
      MAIL_BOX.MSG.INST := ACK_HOST
      MAIL_BOX.MSG.PATHNAME := NULL
      MAIL_BOX.MSG.FILE_SIZE := NULL
      MAIL_BOX.MSG.SUCC_CODE := STORE_COMPLETE
      t := GATEKEEPER.TICKET (MAIL_BOX, C)
      GATEKEEPER.ADVANCE (MAIL_BOX, C)
      GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
    ELSE
      MAIL_BOX.MSG.INST := ACK_HOST
      MAIL_BOX.MSG.PATHNAME := NULL
      MAIL_BOX.MSG.FILE_SIZE := NULL
      MAIL_BOX.MSG.SUCC_CODE := MAIL_BOX.MSG.SUCC_CODE
      !error returned from io process;
       cmd packet received: improper number of data packets!
      t := GATEKEEPER.TICKET (MAIL_BOX, C)
      GATEKEEPER.ADVANCE (MAIL_BOX, C)
      GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
    FI
  ELSE
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
    !file not found; write access to file not permitted!
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  FI
END FM_CMD_HND_STORE_FILE
```

```
FM_CMD_HND_READ_ACL PROCEDURE
ENTRY
  MSG := READ_ACL
  DIR_CNTRL_DIRECTORY (MSG
                       USERID
                       PATHNAME
                       NULL !file type!
                       NULL !access level!
                       NULL !link!
                       NULL) !acl_entry!
  !returns dir_succ_code!
  IF DIR_SUCC_CODE = TRUE
    THEN
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ACL_READ_COMPLETE
    !acl data transfered from acl_buffer;
     host acknowledgement sent!
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  ELSE
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
    !file not found; read access to directory file
     not permitted!
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  FI
END FM_CMD_HND_READ_ACL
```

```
FM_CMD_HND_ADD_ACL_ENTRY PROCEDURE
ENTRY
  MSG := ADD_ACL_ENTRY
  DIR_CNTRL_DIRECTORY (MSG
                       USERID
                       PATHNAME
                       NULL !file type!
                       NULL !access level!
                       NULL !link!
                       ACL_ENTRY)
   !returns dir_succ_code!
   IF DIR_SUCC_CODE = TRUE
     THEN
     MAIL_BOX.MSG.INST := ACK_HOST
     MAIL_BOX.MSG.PATHNAME := NULL
     MAIL_BOX.MSG.FILE_SIZE := NULL
     MAIL_BOX.MSG.SUCC_CODE := ACL_ENTRY_ADDED
     t := GATEKEEPER.TICKET (MAIL_BOX, C)
     GATEKEEPER.ADVANCE (MAIL_BOX, C)
     GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
    ELSE
     MAIL_BOX.MSG.INST := ACK_HOST
     MAIL_BOX.MSG.PATHNAME := NULL
     MAIL_BOX.MSG.FILE_SIZE := NULL
     MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
     !file not found; write access to directory not
      permitted; acl_entry "pool" empty!
     t := GATEKEEPER.TICKET (MAIL_BOX, C)
     GATEKEEPER.ADVANCE (MAIL_BOX, C)
     GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
   FI
  END FM_CMD_HND_ADD_ACL_ENTRY
```

```
FM_CMD_HND_DELETE_ACL_ENTRY PROCEDURE
ENTRY
  MSG := DELETE_ACL_ENTRY
  DIR_CNTRL_DIRECTORY (MSG
                       USERID
                       PATHNAME
                       NULL !file_type!
                       NULL !access_level!
                       NULL !link!
                       ACL_ENTRY)
  !returns dir_succ_code!
  IF DIR_SUCC_CODE = TRUE
    THEN
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ACL_ENTRY_DELETED
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  ELSE
    MAIL_BOX.MSG.INST := ACK_HOST
    MAIL_BOX.MSG.PATHNAME := NULL
    MAIL_BOX.MSG.FILE_SIZE := NULL
    MAIL_BOX.MSG.SUCC_CODE := ERROR_CODE (DIR_SUCC_CODE)
    !file not found; write access to directory not
     permitted!
    t := GATEKEEPER.TICKET (MAIL_BOX, C)
    GATEKEEPER.ADVANCE (MAIL_BOX, C)
    GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
  FI
END FM_CMD_HND_DELETE_ACL_ENTRY
```

```
FM_CMD_HND_ABORT PROCEDURE
ENTRY
   MSG := ABORT
   DIF_CNTRL_UPDATE (MSG
                     USERID
                     PATHNAME)
   !store cmd needs to free temporary file!
   MAIL_BOX.MSG.INST := ACK_HOST
   MAIL_BOX.MSG.PATHNAME := NULL
   MAIL_BOX.MSG.FILE_SIZE := NULL
   MAIL_BOX.MSG.SUCC_CODE := CMD_ABORTED
   t := GATEKEEPER.TICKET (MAIL_BOX, C)
   GATEKEEPER.ADVANCE (MAIL_BOX, C)
   GATEKEEPER.AWAIT (MAIL_BOX, C, (t+2))
END FM_CMD_HND_ABORT

END FM_COMMAND_HANDLER
```

```
IO_COMMAND_HANDLER MODULE

EXTERNAL

PK_HND_STORE PROCEDURE (SEG_#       LWORD
                        SIZE        LWORD) !number of bits!
  RETURNS (PK_SUCC_CODE             BYTE)

PK_HND_SEND PROCEDURE (SEG_#         LWORD
                       SIZE          LWORD) !number of bits!
  RETURNS (PK_SUCC_CODE              BYTE)

PK_HND_ACK_HOST PROCEDURE (MSG      BYTE)

FILE_HND_SEND_FILE PROCEDURE (PATHNAME      STRING
  RETURNS (FILE_SUCC_CODE                   BYTE)

FILE_HND_STORE_FILE PROCEDURE (PATHNAME     STRING)
  RETURNS (FILE_SUCC_CODE                   BYTE)

INTERNAL

IO_CMD_HND PROCEDURE
ENTRY
  t := TICKET (MAIL_BOX. C)
  AWAIT  MAIL_BOX. C, (t+1))
  DO
    IF MAIL_BOX.MSG.INST
      CASE READ_CMD THEN PK_HND_READ_CMD
      CASE ACK_HOST THEN PK_HND_ACK_HOST
                         (MAIL_BOX.MSG.SUCC_CODE)
      CASE SEND_FILE THEN FILE_HND_READ_FILE
                         (MAIL_BOX.MSG.PATHNAME
                         (MAIL_BOX.MSG.FILE_SIZE)
      CASE STORE_FILE THEN FILE_HND_STORE_FILE
                           (MAIL_BOX.MSG.PATHNAME
                            MAIL_BOX.MSG.FILE_SIZE)
    FI
    t := TICKET (MAIL_BOX, C)
    ADVANCE (MAIL_BOX, C)
    AWAIT (MAIL_BOX, C, ((t+2)
  OD
END IO_CMD_HND

END IO_COMMAND_HANDLER
```

LIST OF REFERENCES

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

COURTOIS, P. J., Heymans, F., and Parnas, D. L., "Concurrent Control with "Readers" and "Writers"," Communications of the ACM, v.14 no.5 p.667-668, October 1971.

DAVIES, D. W., and others, Computer Networks and Their Protocols, John Wiley & Sons, 1979.

DEFENSE Communications Agency NIC 7124, Arpanet Protocol Handbook, by Network Information Center, January 1978.

DENNING(1), D. E., "A Lattice Model of Secure Information Flow," Communications of the ACM, v. 19 p. 236-242, May 1976.

DENNING(2), D. E. and Denning, P. J., "Data Security," ACM Computing Surveys, v. 11 no. 3, p. 227-242, May 1976.

DIGITAL Research, CP/M Interface Guide, 1978.

DIJKSTRA(1), E. W., "The Structure of 'The' Multiprogramming System," Communications of the ACM, v. 11 no. 5, p. 341-346, May 1968.

DIJKSTRA(2), E. W., "The Humble Programmer," Communications of the ACM, v. 15 no. 10, p. 859-866, October 1972.

HABERMANN, A. N., Introduction to Operating System Design, Science Research Associates, Inc., 1976.

HAMMING, D. F., Coding and Information Theory, Prentice Hall, Inc., 1980.

HANSON, B., Operating System Design, Printice-Hall, Inc., 1973.

HONEYWELL, The Multics Virtual Memory, June 1972.

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

MORRIS, R. and Thompson, K., Password Security, A Case History, paper presented to Bell Labratories, April 3, 1978.

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

O'CONNELL, J. S., and Richardson, L. D., Distributed Secure

Design for a Multi-microprocessor Operating System, MS Thesis, Naval Postgraduate School, June 1979.

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

REED, D. P., and Kanodia, R. K., "Synchronization with Eventcounts and Sequencers," Communications of the ACM, v. 22 no. 2 p. 115-124, February 1979.

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

SCHELL(1), Lt.Col. R. R., "Computer Security: The Achilles' Heel of the Electronic Air Force?," Air University Review, v.XXX no. 2, January 1979.

SCHELL(2), Lt.Col. R. R., Security Kernels: A Methodical Design of System Security, USE Technical Papers (Spring Conference, 1979), pp 245-257, March 1979.

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

SHAW, A. C., The Logical Design of Operating Systems, Prentice Hall, Inc., 1974.

SNOOK, T., and others, Report on the Programming Language PLZ/SYS, Springer-Verlag, 1978.

ZILOG(1), Inc., An introduction to the Z8010 MMU Memory Management Unit, Tutorial Information, August 1979.

ZILOG(2), Inc., Z8001 CPU, Priliminary Product Specification, March 1979.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
4.	Lt.Col. R. R. Schell, USAF, Code52Sj 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, USA Box 426 U.S. Army War College Carlisle, Pennsylvania 17013	1
8.	Lt. E. J. Parks, USN NAVSPECWARGRU TWO N.A.B. Little Creek Norfolk, Virginia 23521	2
9.	Lt. C. A. Davis, USN NARDAC San Francisco NAS Alameda Alameda, California 94501	1