IMPLEMENTATION OF A PARALLEL MULTILEVEL SECURE PROCESS

by

David R. Pratt

June 1988

Thesis Advisor: Joseph S. Stewart

Approved for public release; distribution is unlimited
IMPLEMENTATION OF A PARALLEL MULTILEVEL SECURE PROCESS

This thesis demonstrates an implementation of a parallel multilevel secure process. This is done within the framework of an electronic mail system. Security is implemented by GEMSOS, the operating system of the Gemini Trusted Computer Base. A brief history of computer secrecy is followed by a discussion of security kernels. Eventcounts and sequences are used to provide concurrency control and are covered in detail. The specifications for the system are based upon the requirements for a Headquarters of a hypothetical Marine Battalion in garrison.
Implementation of a Parallel Multilevel Secure Process

by

David R. Pratt
First Lieutenant, United States Marine Corps
B.S.E., Duke University, 1983

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1988

Author: David R. Pratt

Approved by: Joseph S. Stewart, Thesis Advisor
              Richard A. Adams, Second Reader
              Robert B. McGhee, Acting Chairman,
              Department of Computer Science

James M. Fremgen,
Acting Dean of Information and Policy Sciences
ABSTRACT

This thesis demonstrates an implementation of a parallel multilevel secure process. This is done within the framework of an electronic mail system. Security is implemented GEMSOS, the operating system of the Gemini Trusted Computer Base. A brief history of computer secrecy is followed by a discussion of security kernels. Eventcounts and sequences are used to provide concurrency control and are covered in detail. The specifications for the system are based upon the requirements for a Headquarters of a hypothetical Marine Battalion in garrison.
# Table of Contents

## I. Introduction  
A. The Environment  
B. Goal  
C. Organization  

## II. Introduction to Secure Computers  
A. History  
B. Covert Channels  
C. Security Kernels  
D. Secure Computers  
E. The Gemini Computer and GEMSOS  
F. Other Secure Systems  

## III. Eventcounts and Sequencers  
A. Eventcounts  
B. Sequencers  
C. Relation to Semaphores  
D. Security of Eventcounts and Sequencers  
E. Implementation of Eventcounts and Sequencers in GEMSOS  

## IV. Research Model and Implementation  
A. Introduction  
B. Design Limitations  
C. Description of Need  
D. Requirements  
E. Overview of the Secure Mail System Design  

iv
I. INTRODUCTION

Imagine yourself as a critically sick patient who has just checked into a large Eastern Medical Center. Right before undergoing surgery, your doctor sits down and logs onto the computer to review your record. The only problem is that your record is nowhere to be found, and there is no trace of what happened to it. A computer "virus" has attacked the system. After rebuilding your medical record, getting the medical care that you needed and recovering, you finally return to work around Christmas time. As you log on to the computer to check the messages that have piled up, you notice that the system is severely overloaded. A type of computer chain-letter has taken over the system in only a couple of hours. Both of these events actually have happened. [Marb88]

One of the problems with the use of computers is the lack of security built into them. It is fairly easy to control the flow of classified paper, but how is the classified computer data safeguarded? To the Department of Defense (DoD), this presents a real problem, and one that is drawing increasingly close scrutiny.

Traditionally, computer security has been an afterthought to system designers whose main concerns are the efficiency of operation and the budget. This attitude has
degraded the performance of systems, caused increased costs, and systems to be delivered late when the security module had to be added. If the designers of the systems had considered the security requirements in the beginning, most of these problems could have been avoided. [Tayl88]

A. THE ENVIRONMENT

Computer security is the art of compromise. The only truly secure computer is one that is turned off and in a locked and shielded room. The computer which is easiest to use places no limits on the activities of a user, authorized or not. The most efficient computer, in terms of throughput, does no checking for authorization to do an operation, it just does it. It is the system designer's job to hammer out a compromise between security, usability and efficiency.

The security policy of an organization determines how much security is compromised in order to achieve more efficient operation of the system. The security policy is determined by the security requirements of the organization. If the principle function of the organization is to count sheets and pillow cases, usability and throughput are more important than security. At an intelligence center the opposite will be true. While these are the two extremes (therefore easily lending themselves to a particular security policy implementation), what about the average military installation? It is harder to develop and maintain
a coherent computer security policy when the users and the designers of the system themselves do not have a firm grasp on the security requirements of the organization. It is those systems that are provided to indecisive and often unenlightened users that give computer security a bad name. The reason is that very often security has been an afterthought and usually, therefore, inefficient, cumbersome, resented, and widely ignored.

The security goal of all computer systems is to provide access to authorized users while denying access to unauthorized users. A secure system is only secure with respect to the security policy of the organization. DoD defines a secure system as one that:

...will control, through use of specific security features, access to information such that only properly authorized individuals, or processes operating on their behalf, will have access to read, write, create, or delete information. [DoDs85]

While this definition gives us an idea what a secure system is, it does not say how one is to be implemented. A more rigorous definition of a secure computer will be developed in the next chapter.

B. GOAL

It is the goal of this thesis to combine the security aspect of computers with the growing field of parallel processing; having two or more programs running at the same time and communicating among themselves. The security will have to allow for a multilevel secure process. A multilevel
secure process is a program that interacts with the security policy at different levels and does not violate it. Since security was a prime consideration from the beginning of the development process, the parallel multilevel secure process is both easy to use and efficient. Each of the processes is a simulated electronic mail network node. The system will simulate a network running on a trusted computer base. This served to investigate the use of interprocess communications and trusted computers in a multilevel secure environment.

C. ORGANIZATION

Chapter II presents a brief rational for the consideration of the systems approach to security of a system. The first section of the chapter is a brief history of computer security. Also in the second chapter the concepts of "Secure Computer" and "Security Kernel" are introduced. The next section is an overview of the Gemini trusted computer base and GEMSOS, its operating system. This section is a condensation of the "System Overview" published by Gemini computers. The final section of the chapter contains some information on other secure systems, both implemented and theoretical.

Chapter III deals with eventcounts and sequences. These provide a means for processes to communicate with each other during execution. The Gemini computer implements these in a secure environment which is a little more complex than normal.
Chapter IV provides a description of the model and implementation of the secure mail system. A complete description of the modules is provided, along with justification for some of the implementation choices that were made during development.

The final chapter, Chapter V, lists the conclusions and provides recommendations for further study and research.
II. INTRODUCTION TO SECURE COMPUTERS

A. HISTORY

When Charles Babbage first developed his mechanical analytical engine in the early nineteenth century, it was a single user--single process machine. Security meant simply keeping it locked up away from physical harm. One person could run only one process at a time and that person had to be in the same physical location as the computer to use it, so physical protection was all that was required.

As technology advanced, the data processed on computers became more sensitive and the cost of the machines increased, the means of physical protection became more elaborate. The fact remained, however, that the only protection needed to enforce computer security was physical. Very little thought went into having to place a security device within the computer itself. To this day the primary emphasis in security remains physical; that is, if you cannot get to the computer, you can not do any harm to the machine or the information stored in it. This eliminates most of the security threats from outsiders. The threat from insiders, people who are authorized access to the computer, remains.

Computers were developed which were able to do more than one process at a time, multiprocessing allowed the user to
have several processes loaded on the system, although only one process was executing at any given moment. The advent of multiprocessing meant that several users could gain access to the system and run different programs at the same time. This fact, combined with the use of remote peripherals (which allowed users who were not in the same physical location as the computer to use it), created the need for a new means of access control. These two developments took the computer out of the exclusive control, both physically and operationally, of a few trusted operators and gave rise to the need for additional security measures. It was no longer sufficient to provide physical security; a form of logical security was needed. Changes had to be made in operating system design to include a means of verifying the person logging on had authorization to access the system.

The user name and password mechanism was designed to allow only certain people, i.e., the "authorized users," access to the system. When a user wants to gain access to the machine, he has to first input his user identification (userid) and a secret password that is known only to himself and the System Security Manager (SSM). These two entries are verified in a table of authorized users. If they are found and are correct, access is granted; if not, access is denied.
The userid and password system can be extended to include not only the computer, but also to certain sets of programs and data files within the computer. This system limits direct access by the user to only those items to which he has been granted access authorization by the SSM. These systems are not fool-proof, however. There is a set of utilities, such as text editors and file managers, to which all users have access. One of these programs could be modified such that when it executed, it would copy a legally accessed, protected data file into an unauthorized and unprotected data file. This type of modification to a program creates what is known as a "Trojan Horse" program. In addition to its intended function, such a program performs unauthorized hidden functions, usually undetected. [Beob85]

A classic Trojan Horse program was written around 1976 at Heriot-Watt University, United Kingdom. A student wrote a program that simulated a system crash followed by a login sequence. He then left the program out on the system for other users to try and run. When the unsuspecting user ran the program the system appeared to crash and the user then signed on. The program recorded the userid and password in a disk file for later use by the author. [Norm83]

A derivation of the Trojan Horse is the "virus" program. This program functions such that every time the user executes the program in which the virus is embedded, the
virus is able to embed itself in yet another program until the entire system is infected [Beob85]. An example of this phenomenon would be a program that appends itself to the end of another program and in turn deletes the program in which it is embedded. Eventually, all of the programs will have been infected and deleted.

The user identification and password system can do nothing to stop these two problems. Since the "authorized user" is the one running the infected program, his actions are entirely legal—the results of his actions, even though they may be unintentional and unknown to him, are not legal or authorized. To combat the use of "Trojan Horses" and "viruses," a new method of computer resource security had to be developed. [Beob85]

The concept of multitasking of the computer created a special kind of problem, namely, "How to separate two processes that require different levels of security?" For the most part, this was handled by limiting the system to one classification at a time, the so called "single level" security. Whenever the classification of the jobs changes the machine has to be purged of all data to ensure there is no residual classified information left on the machine. One of the major drawbacks to this system is that one user processing a classified job will cause all unclassified jobs to wait until the classified job is done and the system has
been sanitized. This was clearly a waste of computer resources.

As part of the research to deal with these security issues, the concept of a "Security Kernel" (hereafter referred to as just kernel), was developed. This concept is the main focus of this thesis. As the DoD becomes more and more computerized, emphasis must be placed on the security aspects of computer systems during the entire system life cycle. Computer security cannot simply be added as an afterthought software package.

B. COVERT CHANNELS

The Trojan Horse programs require a means to transfer information from the authorized user to the perpetrator's desired destination. Most of these information paths can be closed, or reduced, by the use of the reference monitor [Ames73]. However, there are ways to transfer information from one process to another that do not use normal data transfer means. This leakage of data between programs that use data paths not intended for information transfer are called covert channels [Lamp73]. All computer systems have an abundance of covert channels, it is only the secure systems that are concerned with eliminating them.

The damage done by the channel is a function of its bandwidth. The bandwidth is the measure of bits per unit time that are passed though the channel per unit time. The
higher the bandwidth, the more damaging the channel is to the security policy of the system.

While there are many different specific types of covert channels, they can be grouped into two general classes, storage and timing channels. A storage channel is one that causes an object to be written and another process can observe some aspect of that action. A timing channel uses a timing mechanism to observe the effect on the system by some process. [Gass88]

Storage channels can be grouped into three subclasses. The first class is the object's existence. This simply tells the user if an object exists or not. An example would be an attempted access to a file and the message "permission denied" is returned by the system. In this manner we can tell the file exists. The second type, object attributes, can give us even more specific data on the object. This can be done by reading an object's header and reading the attributes. The value of attributes that are stored in the header may be real or placed there by a Trojan Horse and used for communication. The final type of storage channel is the shared resource channel. This channel communicate more on the status of the system rather then on one particular process. A printer spooler that has a finite number of jobs can be monitored as a covert channel; this would indicate the status of the print queue at any given moment. [Gass88]
The other general type of covert channel is the timing channel. This type of channel requires access to a timer in order to operate. The clock can be provided by the system, i.e., a real time clock, or by the program, i.e., a timing loop. From the passage of time it is possible for the program to determine the passage between two events. An example of this is the request for access to a file and denial of access. The programmer knows that it takes X amount of time for the system to determine that it does not exist and Y to determine that access is denied. [Gass88]

Of the two types of channels, the timing channel is harder to control. There are no formal techniques for finding them and they are very difficult to detect and correct. The storage channel's bandwidth can be reduced by strictly enforcing the security models and the elimination of shared resources. [Gass88]

C. SECURITY KERNELS

As the problem of covert channels was brought to light, a method to deal with them had to be developed. The most elementary solution was to provide a separate machine for every level, or security classification, of processing. This was also one of the most expensive solutions since the computer was not being used to the fullest extent possible and it was difficult to share the data between machines. This idea evolved into the concept of having the machine
appear to each user as though it were dedicated to his particular level of processing.

This concept required the establishment of several different security levels within the machine itself. Providing these various levels of security were extensions of the password and userid system. The user had his access authorization checked at a finer level, thereby adding an extra layer of security to the system. An example of this is requiring the user to specify a password to access an object. This method of access control proved to have the same drawbacks as the login password--it created an environment, although smaller, in which the user access could be exploited. As was shown in a preceding section, access control of the environment can be circumvented by the Trojan horse or virus.

The environment created by use of access controls provides only a means to check the user's authorization to access the data, which is insufficient to stop the Trojan horse attack. We must also examine his authorization to modify, delete and write to the data storage location. In order to reduce the bandwidth of the covert channels, the authorization to write and the destination of the data must be validated each time the user writes his data. Simply put, every reference to any information must be checked and authorized. This is the basic concept behind the idea of a reference monitor as shown in Figure 1 [Ames73].
Before we go any further, some terms have to be defined that will be used throughout the rest of this thesis. All active processes, be they users, executing jobs, or anything else which makes a reference to data, are termed subjects. An object is a passive element, such as a data file, program file, terminal device, or storage device, which contains the data elements of the system. When a program is called, it transitions from an object—a passive program file, to a subject—an active process in the system.

Nondiscretionary security is the mandatory security that is enforced on all users. It is based strictly on the individual's security clearance. Discretionary security is the policy that limits access to those who have the need to know. A security policy is the organization's guiding principle when it comes to accessing information. This can be discretionary—relying strictly on the subject's need to know—or nondiscretionary—based on the subject's level of
trust, which is his security clearance. Most organizations, like the U.S. military, have a policy that is a combination of both of these. A clearance and a need to know are both required to gain access to objects. With reference to this security policy, we can classify computers as trusted or not. A trusted computer is one which can be relied on to enforce the organization's security policy.

When a subject references an object, the reference monitor must approve the transaction. This includes not only reading the data in a file, but writing the data out as well. Note that this reduces the effectiveness of the storage covert channels by controlling all access to the data. The Trojan Horse program is detected when it tries to write the data into an unauthorized file, and the virus is diagnosed when it attempts to embed itself where it is not allowed.

Some of the more successful implementations of a trusted computer use the security kernel [Land73]. The security kernel is defined as the hardware and software required to carry out the reference monitor concept. The kernel assumes control over a small subset of the functions which are normally part of the operating system. [Ames73]

The kernel is placed between the operating system and the hardware. The implementation of the kernel is of vital concern to the design of the system. Since every data reference must be validated, a significant amount of
computer time is spent in the kernel. If the kernel is implemented in software the performance will suffer, but the system will be flexible. A hardware kernel will run fast but it will be very difficult to modify. As stated in the introduction, security is the art of compromise. As a result, the kernel should be implemented partially in both. Figure 2 shows a normal system configuration and Figure 3 shows a system which has a security kernel.

<table>
<thead>
<tr>
<th>Users</th>
<th>User Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applications</td>
<td>Operating System Interface</td>
</tr>
<tr>
<td>Operating Systems</td>
<td>Hardware Interface</td>
</tr>
<tr>
<td>Hardware</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2. A Standard Operating System

A trusted process, as shown in Figure 3, is one that can circumvent the security built into the kernel. While it can sidestep the built in security, it is trusted not to violate the organization's security policy. This type of process is
critical to the efficient operation of the system. A typical trusted process allows the SSM to down-grade a classified file.

Implementing the kernel as a subset of the operating system solves the problem of size and complexity associated with large programs. This implementation concept is integral to the three design criteria of secure computers (completeness, isolation, and verifiability). The size of the kernel has a direct impact on the designer's ability to prove that each of the design criteria hold.

Figure 3. An Operating System using a Security Kernel
The first of these, completeness of the reference monitor—requires that all access to the objects be made through the kernel. The second concept, the isolation of the kernel, ensures that the monitor is tamper-proof. Isolation of the kernel is usually achieved by implementing the monitor in a mixture of hardware and inaccessible system software. The third concept is that of verifiability of the reference monitor. This requirement states that the designer of the system must be able to prove that the monitor enforces the security policy for which it was designed.

The completeness and verifiability of the reference monitor can be attributed to small size of the kernel. Due to the small size, it is possible to do exhaustive testing and proof of correctness to prove the correctness of the kernel. An example of the small size of the kernel would be one of the first security kernels developed by The Mitre Corporation in 1974. It consisted of less than 20 primitive subroutines and was written in fewer than 1000 high level language statements. [Ames73]

The work in security kernels was based mostly on the development of the Bell and LaPadula model [Ames73]. This model is the most widely accepted of the systems that have been built thus far for use within DoD. The model is based on the "simple security condition" in which a subject at a given security level has the ability to access only objects
at an equal or lower security level. Objects of a higher classification would be inaccessible; in other words, no "read up" is possible.

The **-property (pronounced "star-property"), is just the opposite of the simple security condition. Subjects can only write to objects that are higher or equal classification, no "write down" is allowed. Figure 4 contains a graphical representation of the Bell and LaPadula model.

Figure 4. The Bell and LaPadula Model

An exception to the two properties is a trusted process, in which subjects are authorized to cross some of the security boundaries of the system provided that the security policy of the system is not violated.
D. SECURE COMPUTERS

The DoD realized there must be some standardization in the definitions and criteria of secure computers. As a result of this the DoD Computer Security Center (DoDSCS) published the DoD Trusted Computer System Evaluation Criteria, CSC-STD-001-83, otherwise known as the "Orange Book" (the color of its cover) [DoDs85]. This document sets forth six fundamental requirements that a trusted, or secure, computer must provide. In addition to the requirements, four divisions and several subclasses are defined to provide a standard bench mark for the evaluation and rating of the systems. [DoDS85]

The six requirements are broken down into two categories. The first, which contains the first two requirements, deals with the policy that is being implemented. The remaining four requirements cover what the system must furnish to ensure controlled access to data. The following is a summary of the requirements:

Policy Requirements

Requirement One. Security Policy--there must be an explicit and well-defined security policy enforced by the system.

Requirement Two. Marking--Access Control Labels must be associated with objects.

Accountability Requirements

Requirement three. Identification--Individual subjects must be identified.
Requirement Four. Accountability--Audit information must be selectively kept and protected so that actions affecting security can be traced to the responsible party.

Requirement Five. Assurance--the computer system must contain hardware/software mechanisms that can be independently evaluated to provide sufficient assurance that the system enforces requirements one through four above.

Requirement Six. Continuous Protection--the trusted mechanisms that enforce these basic requirements must be continuously protected against tampering and/or unauthorized changes. [DoDS85]

These six basic requirements provide the foundation of the four security divisions. The divisions are labeled alphabetically, in decreasing order of assurance of the security enforcement, D being the least credible and A providing the most complete security mechanisms. Each class includes all of the requirements for the lower classes.

Division D has only one class. A division D machine is one that has been tested but has failed to meet any the requirements of a higher class. Minimal protection is provided by this class.

Discretionary protection is provided by both class C1 and C2. The users, processes and other active entities are held accountable for the actions by required audit capabilities. A C1 system must control access between named users and named objects. The users of the system shall be able to specify and control the access to an object. Before a user gains access to the system he must identify himself and authenticate his identity. All functions of the TCB must be protected from tampering and be able to be
periodically validated to ensure correct operation. The Cl system shall be documented and tested to ensure that the documentation agrees with the implementation. A Cl system should only be used when all users are processing the same level of data.

A class C2 system has a finer granularity on the discretionary access control than C1, because it holds the individual responsible for his actions by means of login procedures, auditing of security-relevant events and the isolation of resources. When a system assigns a storage resource it must first verify that the unauthorized data has been purged. The testing process for a class C2 system must include a search for obvious security related flaws in the system.

Division B, mandatory protection, has the largest number (three) of classes. Division B enforces a set of mandatory access controls through the use of sensitivity labels that are associated with the data in the system. The security labels include both machine and human readable formats. The developer of the system must be able to provide the specification of the system and prove that the TCB implements the reference monitor concept. A TCB that has been rated as class B1 must provide an informal statement of security policy model, data labeling, and mandatory access control over named entities in the system. Any change to the security labels or overrides of the system must be
auditable and done by an accountable individual. Any known bugs in the system must be removed before certification of the system. Documentation must be provided that includes the maintenance and user changes to the TCB.

A B2 system requires that the security policy be formalized and extended to include both discretionary and nondiscretionary controls over all entities of the system. The system must provide a trusted path from the user to the TCB for user login and authentication. All physical devices on a B2 system must have a minimum and maximum security level. The process isolation requirement for class B2 requires that each process contains its own address space under TCB control. A detailed search for covert channels is mandated in a B2 system and the bandwidth of the channel must be computed. The TCB must be structured in such a way as to provide protection critical and nonprotection critical elements in the system. A configuration management system must be put in place to ensure consistency between the TCB and the documentation. The developer of a B2 system must ensure that the formal model used and defined in the descriptive top-level specification is consistent with the TCB.

A system that has achieved a B3 classification makes use of security domains to aid in its high resistance to penetration. The B3 rated TCB must completely implement the reference monitor concept, be tamperproof and small enough
to be thoroughly analyzed and tested. A logically isolated 
and distinguishable trusted path must be provided between 
the user and the TCB which can be activated by the user or 
the TCB. In the event of system failure, a means to provide 
a trusted recovery, one that does not compromise the 
security of the system, must be in place. The coding of a 
B3 TCB must be done using modern software engineering 
techniques. The testing of the TCB must find no design 
flaws, show that few correctable implementation flaws exist 
and that there is cause to believe that few flaws remain. 

Current technology allows for only one class within the 
A division, A1. While the system may be functionally the 
same as a class B3 system, the amount of analysis, formal 
design specifications, and verification methods result in a 
high degree of credibility that the TCB is correctly 
implemented and that the hardware implements the formal 
specification. The formal specification must contain a top 
level specification of each of the modules in the model, a 
formal model of the security policy, and a mathematical 
proof proving its correctness. The existence of covert 
channels must be identified, analyzed, and their existence 
justified by formal means. 

The purpose of the Trusted Computer System Evaluation 
Criteria is to provide guidance to both the user and the 
vendor. It is a service to the user by aiding in the 
acquisition process. By having a reference document, the
user can specify a class of protection that he needs, thereby eliminating the need for the development of his own security classification system. The document provides a service to the vendor by listing those requirements the government views as important. It also gives the vendor the evaluation criteria so that he can design and build systems that will have a market.

E. THE GEMINI COMPUTER AND GEMSOS

This section serves as a brief overview of the Gemini Trusted Multiple Microcomputer Base, hereafter referred to as the computer, TCB or system. This is essentially a synopsis of the salient points contained in [Gemi84], which is available from Gemini computers.

The system was designed from the ground up to be certified as a B3 class machine with the possibility of eventual A1 rating. In order to accomplish this, the system uses some of the latest microprocessor and software technology. Some of the major features of the system include:

1. Use of the Intel IEEE standard 796 Multibus allowing for third party expansion boards.
2. Up to eight iAPX286 (80286) microprocessors with up to two megabytes of local memory.
3. Global shared memory of up to eight megabytes.
4. Nonvolatile memory used to store passwords, encryption keys and other security related data.
5. Up to 48 RS-232 serial communication ports.
6. A mix of four disk drives to include Winchester hard disks and floppy disk drives.

7. Real time calendar clock.

8. Self-hosting software development environment.

9. Data encryption using the NBS standard DES algorithm. [Gem85]

A graphic representation of the system's architecture is contained in Figure 5. The design of the computer provides for a flexible and expandable system capable of growth and customization to the desired application.

1. Resource Management

One of the major functions of any computer's operating system is that of resource management. The Gemini Secure Operating System (GEMSOS) is no exception to this rule. GEMSOS is structured as a kernelized operating system, and as such the system calls are made as procedural calls to the kernel. By providing a conceptually simpler operating system, the resource management calls have been divided into three major areas: segments, process and device management. The specifics of the individual calls can be found in the GEMSOS interface routines provided by Gemini Computers with each compiler; we will deal only in a high level of abstraction.

GEMSOS does not use files as thought of in a conventional sense, but rather makes use of a uniquely identified logical object called a segment. All code and data are contained in a separate segment. By separating
the code from the data segments it is possible to ensure the static nature of the code by making the code segment read only. A pair of functions allows the system to assign a
local temporary identification to a segment and then to release it. Through the use of the "swap-in" and "swap-out" kernel calls, it is possible to bring a segment into memory where the data can be accessed. Secondary storage (disk drives) is divided into a series of volumes. Each of the volumes can be thought of as a collection of segments. The volumes, just like the segments, contain security labels that reflect the security classification of the data stored in them. The database of segments is managed by a segment manager which keeps track of all segments known to a process though the use of a "Known Segment Table." It is this segment manager that acts as the reference monitor by controlling data access. More specifics about the kernel calls used in this thesis can be found in [Gemi86b].

Process management is the second major area of concern. Most modern computer systems are capable of supporting multiprogramming (having more than one job in memory at a time on a single CPU) and multiprocessing (executing more than one process at the same time on multiple CPUs); the Gemini system is no exception. GEMSOS requires that the processes are run on the same physical CPU that they are created on at run time. This forces the process to share the CPU with other executing processes. To minimize bus contention, each process's code, stack, and data segments are loaded in the processor's local memory thereby improving the system throughput. In order for two
asynchronous processes to communicate with each other, Reed and Kanodia's eventcounts and sequencers [Reed79] are used (this will be covered in more detail in the next chapter). The time sharing of the CPU uses a very simple algorithm: a process runs until it blocks, at which time it is swapped out and the next pending process is swapped in. In order to keep the kernel code as simple as possible, no effort is made to determine if deadlock exists.

The desire to keep the kernel code as small as possible led to the philosophy for device management that Gemini used in designing the system. This approach is to handle each I/O function at the application level by the application programmer, thereby making use of part of the segment and process manager subsystems. While this approach makes the verification of the security system easier, it makes the development of application programs considerably more difficult than in a "normal" programming environment. The device management system is based on the requirement that each of the I/O peripheral controllers are themselves processes. These processes are activated by a procedural call at the application level and accomplish the required I/O synchronization and transfer at a lower level. The involvement of the kernel with I/O is minimal. It is limited to the attachment and detachment of the device to the process, which makes it possible to reduce the amount of
involvement of the kernel in the process, thereby increasing throughput.

The resource management within GEMSOS is highly dependent on the hardware of the system. This follows directly from the fact the system was designed from the ground up to be a secure system.

2. **GEMSOS Architecture**

GEMSOS uses a ring-based protection system, similar to the Multics operating system [Corb65]. The rings are referred to as Ring 0, the most privileged, through Ring 3, the least privileged. Rings 0 and 1 implement the Bell-LaPadula model. Ring 0 contains the distributed kernel that implements the nondiscretionary part of the model. Ring 1 contains the supervisor that provides the discretionary part of the model. These first two rings make up the reference monitor. Rings 2 and 3 are outside the security perimeter of the system and are used for nonsecure processes. GEMSOS provides a series of kernel calls to allow a process to communicate across different rings.

Each entity within GEMSOS is assigned a security label. From this label it is possible to determine the level of compromise and integrity properties of the subject or object. Figure 6 contains a brief statement of these two properties as contained in [Gemi84]. When entity A's access class is a superset of entity B's access class, A's access class is said to dominate B's access class.
Compromise Properties:
1) If a subject has "observe" access to an object, the compromise access component of the subject must dominate the compromise access component of the object.
2) If a subject has "modify" access to an object, the compromise access component of the object must dominate the compromise access component of the subject.

Integrity Properties:
1) If a subject has "modify" access to an object, the integrity access component of the subject must dominate the integrity access component of the object.
2) If a subject has "observe" access to an object, the integrity access component of the object must dominate the integrity access component of the subject.

Figure 6. Compromise and Integrity Properties

The access class of the entities determines what type of device they can interact with. This is made more complex by the fact the GEMSOS allows single and multilevel subjects. These are subjects that can access objects over a contiguous range of security levels. This is similar to a multilevel device, one that can be attached to different level subjects. GEMSOS also supports single level devices. Figure 7 list the properties of single and multilevel devices.

3. Application Development

This section contains some of the background and procedures required for a programmer to develop applications within GEMSOS. The steps taken apply to all programming
Single Level Devices:

1) To receive ("read") information:
   Process maximum compromise >= Device minimum compromise
   Device maximum integrity >= Process minimum integrity

2) To send ("write") information:
   Device maximum compromise >= Process minimum compromise
   Process maximum integrity >= Device minimum integrity

Multilevel Devices:

1) To receive ("read") information:
   Process maximum compromise >= Device maximum compromise
   Device minimum integrity >= Process minimum integrity

2) To send ("write") information:
   Device minimum compromise >= Process minimum compromise
   Process maximum integrity = Device maximum integrity

---

Figure 7. Properties of Single/Multi Level Devices

languages supported by the Gemini computer. For further guidance the reader should refer to [Gemi86b] and [Gemi86c].

GEMSOS is capable of hosting an operating system (having another operating system run between GEMSOS and the applications). Currently this is limited to CP/M-86, but discussions with Gemini personnel indicate that GEMSOS might soon be able to host the UNIX operating system as well [Tao88]. The ability to have a hosted, widely-used operating system is critical to the application development process, allowing users to run some of the commercially available programming languages such as Pascal MT+,
JANUS/ADA, PL/1, C, and Fortran. This reduces the amount of code required to be included within GEMSOS by having the hosted operating system handle the development process. There are special routines that are provided by Gemini computers to create the operating system and kernel calls to GEMSOS for the compiled code. These special routines allow the user to write programs which do not require the hosted operating system but can place service calls directly to GEMSOS. These service calls are similar to normal procedural calls for the language in which the application is written.

One of the advantages of having CP/M as a hosted operating system is that GEMSOS allows concurrent processing without depending on concurrent programming languages. The programs can be developed under CP/M and then run in GEMSOS as concurrent programs. For example, PASCAL MT+ does not have the ability to effect interprocess communication but, with functions provided by GEMSOS, it is possible to use the eventcounts and sequencers to achieve the communication.

The coding, compilation, and linking of an application is done in a manner similar to what is done in a standard CP/M environment. The coding is a little more complex because of the security constraints involved. The debugging of the system is radically different in that the system must be sysgened (defined later) and then booted.
under GEMSOS. This by itself adds a tremendous amount of
time to the application development process.

One of the most difficult concepts that the
application developer faces is the structure of the GEMSOS
hierarchical storage system. As stated previously, GEMSOS
does not support a file and directory structure, but rather
a hierarchical segment ordering where each of the segments
has a unique name, its access path. The segment naming
process follows a strict hierarchical method that is shown
in Figure 8. The segment numbers are assigned in a CP/M
submit file. This file is then used as the source input for
the sysgen process, which builds the structure on the
desired volume. The sysgen process is covered in detail in
[Gemi85a].

4. Summary

The Gemini Trusted Multiple Microcomputer Base
provides a flexible, cutting edge of technology computer
system to be used in an environment where security is a key
consideration. While the system is very capable, it is
still a first generation TCB, and like many other products
on the leading edge it is not user friendly. If the
application that is being developed does not require the
security controls provided by the system, use another
machine.
Figure 8. Gemini's Hierarchical Storage Structure
F. OTHER SECURE SYSTEMS

The Gemini TCB was designed from the start to be a secure computer. [Land73] provides a good overview of some other systems that have been completed and some that are still under development. Many of these systems are extensions to existing software or hardware.

Two operating systems seem to be the favorites for the software implementation of the security models, Multics and Unix. Multics [Corb65] is a logical choice for the conversion since its design is based upon the ring privilege concept, the inner rings are more privileged than the outer rings. By establishing a few well-defined gates it is possible to control the flow of information between the rings. The use of segmented memory, where each file is a segment, allows the inclusion of a header to keep track of the ring parameters. Each segment has read, write and execute bits that act in conjunction with the ring parameters to aid in the enforcement of the security policy.

The other popular operating system to enhance is Unix [Ritc74]. In native, or unenhanced Unix the protection system is based on the file system and the user domains. Each of the files has read, write and execute bits for the owner, group, and world. This provides a basic data access security. One of the better known modifications was the UCLA data secure Unix. In this implementation, the Bell-LaPadula model is enforced by a module running outside
the kernel. The resulting system was implemented on a PDP-11 and ran very slowly [Land73].

A different approach was taken by Honeywell for development of the Honeywell Secure Communications Processor (SCOMP) [Hone84]. To build this system, a standard minicomputer, the Honeywell DPS 6, was modified by the replacement of the central processor unit, the memory management unit and the addition of a security protection module. SCOMP uses a kernelized operating system based on Multics. This system has been certified by DoDSCS to meet all the requirements for the A1 level.

Computer security, especially security kernels, was a major research area in academia during the early eighties. As the research started to yield implementable systems, fewer papers were published to avoid giving away trade secrets and benefiting competitors in the security market place.
Whenever a computer system has more than one process executing concurrently, a process management system is required. When the processes are independent of each other, the operating system's scheduler and process swapping mechanism provides the required control. In a computer system that allows the processes to communicate, share code, or share data during execution, a means to achieve process synchronization and communication is required.

There are several means to achieve the synchronization necessary for the correct process execution. Some of the more common methods (semaphores and monitors) are primarily designed to provide mutual exclusion to a critical section of code (only one process can execute at a time) or the access to a data structure. This chapter will explore a different form of synchronizing mechanism that is used by GEMSOS, eventcounts and sequencers. This mechanism to control the sequencing of processes was developed by Reed and Kanodia [Reed79].

A. EVENTCOUNTS

An eventcount is an increasing unbounded integer that keeps track of the number of events that have occurred so far in the system. This concept is very similar to Lamport's "logical clock" [Lamp78]. It is up to the
programmer to determine what constitutes an event; it could be the completion of a procedure, the availability of a computed result, or an error condition. [Reed79]

The eventcount can only be modified by placing a call to the advance(EVC) procedure, where EVC is the eventcount in use. This has the result of increasing the value of EVC by one. By doing this it is possible to signal the system of the occurrence of an event.

The value of an eventcount can be read by the read(EVC) function. This function returns the current value of the eventcount, with the value being the number of advance(EVC) calls that have been placed before the call. Since mutual exclusion is not guaranteed, it is possible that the value of the EVC can be changed during the read operation. This equates to the read function returning the minimum value of the eventcount at any given moment.

Constant reading of an eventcount provides a way to monitor the occurrence of an event. The busy wait loop can be avoided by the use of the await(EVC,x) primitive. The use of this primitive causes the calling process to suspend until the value of EVC is equal to or greater than that of x. If the value of x is less than or equal to the value of the eventcount at the time of the call, the process is not suspended. [Reed79]
B. SEQUENCERS

One of the drawbacks of the use of pure eventcounts is the lack of mediation between concurrent processes that must be synchronized. An example of this is two processes that are trying to update a file at the same time. There has to be some mechanism to guarantee that one request is processed before the other to ensure consistency of the data. Reed and Kanodia [Reed79] describe an additional object called a sequencer, which provides the ability to differentiate between two processes that act independently. It does this by using a ticket(SEQ) primitive, where SEQ is the sequencer.

The ticket(SEQ) function, much like the read(EVC) operation, returns the current value of the sequencer. However, the ticket function has the side effect of incrementing the value of the sequencer by one. This, combined with the use of mutual exclusion for the ticket section of the operating system, which guarantees that only one ticket request will be processed at a time, ensures that for each call, the ticket function will return a unique value. From the value that was returned from the ticket operation it is possible to determine which process requested a ticket first.

To aid in understanding the use of a sequencer in conjunction with an associated eventcount, the bakery ticket machine is often used as an example. In this example the
customer walks up to the machine and takes a ticket. Since only one customer can take a ticket at a time, each ticket value is unique. This is the ticket operation with the ticket machine acting as the sequencer. The customer then sits down and waits until the turn indicator on the wall, the eventcount, reaches his ticket value, the await operation. After the baker finishes with a customer he increments the turn indicator, the advance primitive, and calls for the next customer.

C. RELATION TO SEMAPHORES

An interesting side light is the claim in [Reed79] that semaphores can be built out of eventcounts and sequencers. This is from the view that eventcounts and sequencers are lower level then semaphores. The paper shows how to construct P and V, and even a simultaneous P operation out of eventcounts and sequencers. [Reed79]

The Concurrent Computer Corporation chose eventcounts and sequencers to implement some of the primitives required for a new operating system. In [Rosk86] it shows that it is not always possible to construct semaphores out of eventcount and sequencer, because of the lack of a conditional ticket operation. If a ticket is taken it must be used, or a dummy process must take the place of the original process and advance the eventcount.
D. SECURITY OF EVENTCOUNTS AND SEQUENCERS

Of special interest is the suitability of the primitives to the secure computing environment. The advance operation can be classified as a pure write. In a pure write no information about the value of the eventcount, either current or previous, is transmitted back to the calling process. This property makes it possible to advance an eventcount that has a security classification at the same or higher level of the calling process, the modify domain.

The read and await primitives can be thought of as pure reads, because no information is modified when the values are returned. There is no primitive to determine if other processes are waiting for the eventcount, making it impossible for one process to determine the status of other processes. Thus, the read and await primitives can be used on eventcounts of equal or lower security classification than the calling process, the observe domain.

The ticket operation on the sequencer is both a read/write operation. Since the ticket operation returns and changes the value of the sequencer, the ticket operation can only be used in the intersection of the modify and observe domains. Thus the sequencer must be at the same security level as the calling process.

Using eventcounts, it is possible to introduce a "secure readers-writers problem." The underlying idea is that the readers do not have the ability to modify any of the data in
the database or to signal any of the writers or other readers. [Reed79] provides implementation to solve this problem in its purest sense. Of interest to this thesis is a modification to this problem, the "multilevel secure readers-writers problem." The problem is constructed by adding multilevel security to the "secure readers-writers problem."

F. IMPLEMENTATION OF EVENTCOUNTS AND SEQUENCERS IN GEMSOS

To provide the required process synchronization Gemini Computers chose eventcounts and sequencers. The shared main memory of the Gemini Computer provided the required architecture for the execution of the synchronization mechanism. The built-in security aspects of operations made them the ideal choice for a secure system. The pure read and writes of the primitive operations are considerably simpler to verify than some of the traditional synchronization mechanisms.

One of the goals in designing a security kernel was to keep the kernel as small as possible. In order to do this, GEMSOS views each eventcount and sequencer as an integral part of a segment. The naming and the security classification of the eventcount and sequencer is the same as that of the owning segment. By having common names, the kernel has fewer entities to keep track of for security purposes. While there is wasted space created by unused eventcounts and sequencers, it is more than compensated for
by the reduced kernel size and complexity in the naming of the objects. [Gemi84]
IV. RESEARCH MODEL AND IMPLEMENTATION

A. INTRODUCTION

A software system was created to explore the multilevel secure process and to demonstrate success of the proposed concept.

The system was developed in the framework of an electronic mail system where each user represents a process. This allows for the creation of multilevel secure data which is sent to and used by a multilevel secure process. The system was first be developed to run with two users of the same level and then was extended to different levels and to more users. The implementation was done primarily in Pascal MT+ and on the Gemini Trusted Microcomputer.

B. DESIGN LIMITATIONS

The overriding limitation in the design of this system was the availability of the Gemini TCB. The availability of the hardware forces the design decision later in the development process. As a result of the availability of support documentation and software available from Gemini Computers the Pascal MT+ language was chosen for this implementation.
C. DESCRIPTION OF NEED

An electronic mail network was chosen to model the parallel multilevel secure processes. The electronic mail system was chosen for its inherent parallelism. It is assumed that multiple users might be active at any given moment. The mail system was made multilevel secure to fully exercise the capabilities of the TCB. The implementation of this system is done to prove that, given that combination of hardware and software support and an integrated security design, a multilevel secure process can operate without severe performance degradation.

1. Environment of Employment

For the purpose of illustration a fictitious United States Marine Corps Infantry Battalion headquarters will be used. Figure 9 shows an organizational diagram. For simplicity, assume that the battalion is in garrison and will not take this system to the field.

![Organizational Diagram](image)

Figure 9. Marine Infantry Battalion's Headquarters
The Commanding Officer (CO) would task the Executive Officer (XO) with gathering all the required information on data usage and security requirements to present to the Divisional Information Systems Management Officer (ISMO). The ISMO will then develop the technical specification of the system. The XO and ISMO will then oversee the contracting and installation of the system.

In the battalion the individual sections each have a security requirement based upon the type of data that they deal with in execution of their duties. The Personnel section (S-1) deals with CONFIDENTIAL data which deals with the status of forces and privacy act information. The Intelligence section (S-2) has all the information on battle plans, both friend and foe, which are classified at the TOP SECRET level. The Operations section (S-3) has the TOP SECRET mobilization and deployment plans as well as the schemes of maneuver and weapons data. All of the SECRET data which deals with the status of the supplies and logistics is kept by the Logistics section (S-4). The Chaplain is not authorized access to any of the battalion's classified data, but does need to access unclassified data on the system. Both the CO and XO have to be able to access all data within the battalion, and as such are classified as TOP SECRET users. All of the battalion's sections are cleared only up to and including the level of the data being processed by that section.
2. **Conventional Solution**

Based upon the data and security requirements of the battalion, two different approaches are possible for the implementation of an electronic mail network. The first is the use of four separate electronic mail networks. Each one of the networks would operate at a single level of security, yielding a single level system. Figures 10(a) through (d) show how the sections would be connected to the different networks. In this solution four separate network servers are required. This approach requires that the TOP SECRET users have four terminals, one for each network available to them. SECRET users will have three, CONFIDENTIAL would have two, and the Chaplain will have only one terminal on his desk. This system would require a total of 22 terminals, four servers, and multiple cable runs. To check all of the incoming messages the CO would have to login to the four different networks. Clearly, there has to be a more efficient way of implementing the network.

3. **Multilevel Secure Solution**

A much more efficient use of resources would be to combine the four different levels of security on one machine. This approach is not unique [NRLR82; Wyat84]. The implementation of this system requires the use of a Trusted Computer Base (TCB) to act as the central message server and one terminal at each of the nodes, seven total. The resulting reduction in the amount of hardware required will
Figure 10. Single Level Networks
result in a system that will be easier for the user to employ. The coding of the system will be more complex and the individual pieces of hardware will be more expensive.

D. REQUIREMENTS

The requirements of the model have been broken down into two general categories; user interface and computational. The separation of the two requirement areas allows the division of the Secure Mail System (SMS) into two main logical divisions. The user interface corresponds to the unsecure section, and the computational requirement is fulfilled in the security relevant sections.

1. User Interface Requirements

The user interface of the SMS was designed to provide a simple interactive single screen text processor that the user could master in relatively few sessions. Thus, WordStar-like editing commands were chosen for the basic editing functions. The selection of an option is done from menus or boolean (yes/no response) questions.

After the user has logged onto the system he will be presented with a menu of options and a listing of the current messages. The actions from the main menu will be able to create, edit, delete, read, or send a message. From this menu the user will be able to terminate his current mail session.

The user will be able to select from a menu of up to nine pending messages to edit, delete or send. A similar
list of up to nine incoming messages will be available to read or delete.

The following is a summary listing of the minimal requirements for the SMS message editor:

1. Single screen (22 lines by 80 characters).
2. Heading to indicate destination, classification, and Date and time created on top line of the screen.
3. Full Cursor control movement within the text area.
4. A means to toggle text insert on and off.
5. Line wrap. (When you reach the end of the line the cursor goes to the first position of the next line.)
6. The return key works as expected; position the cursor on the first character of the next line.
7. Must provide a unique key to end the editing of the message.
8. Save/No save option after all creation and editing.
9. Ability to delete the current character, the one the cursor is under.
10. The ability to edit a previously created, but unsent, message.
11. Recall a previously created message for modification, transmission and/or retransmission.

2. Computational Requirements

The computational requirements have been separated from the user requirements to decrease the amount of security relevant code. The security system has been divided into three major subareas; system configuration, user authentication and data access.

System configuration is done by the System Security Manager (SSM) at boot time. This process involves selecting
the terminal ports and the security levels for the selected ports. The selected security level is the maximum security level of data for the terminal, the minimum is set to UNCLASSIFIED by default. The system supports four classification levels, UNCLASSIFIED through TOP SECRET, without any compartments. Once the SSM makes these selections they remain static until the system is rebooted and reconfigured.

User authentication is accomplished by GEMSOS when the system is booted by the SSM and within the SMS when a user tries to log on to the terminal. The login process verifies the user by a login and password combination. It next prompts the user for a desired security level for the session. The user's request is then checked against the user's and terminal's upper security bounds. If the requested classification is out of bounds, the login/password are incorrect, or the user is not authorized access to the system and access is denied without divulging the reason.

Once the user has been admitted to the system GEMSOS handles most of the data access authentication. The exception is when a user desires to send a message to another user. At this point the SMS must verify the receiver has access to data at that security level before passing the write request to GEMSOS for execution.
E. OVERVIEW OF THE SECURE MAIL SYSTEM DESIGN

In this section an overview of the SMS will be presented at the module level. A more detailed description of the procedures can be found in Appendixes B through D which contain the SMS Code. The segment storage structure that is generated by the system generation process (sysgening the system) can be found in Figure 11. The loader and operator login processes used in the system are the standard processes provided by Gemini Computers. Since these two modules are covered in [Gemi86c], they will not be covered in this thesis. The logical relationship between the two SMS processes is shown in Figure 12.

1. Data Structures

To gain a firm grasp on the structure of the SMS, knowledge of the system's data structures is required. It is how these structures are stored and accessed by the machine that affects the security credibility of the system. As with all data that is stored by GEMSOS, each of the data structures, has a security level label associated with the segment that contains the data. As stated earlier, each of the segments contains an eventcount and sequencer that is maintained by GEMSOS. Through the use of these two mechanisms it is possible to control access to the segments.

The first data structure that is of concern is the user array. This is one of two data structures that is passed from the System Configuration Module to the
Figure 11. SMS Segment Storage Structure
individual node processes (the other being the process definition structure that is required by GEMSOS to initialize a child process). The purpose of this is to provide all of the node processes a listing of all users of the system with the maximum access class, password and user number. This is used by the message sending module to verify that a user can receive mail at the desired security classification level. The user login module reads this array to ensure that a person trying to log into the system is an authorized user. This array is created prior to sysgening the system and remains static during the operation of the system. This allows the SMS to service users that do not have access to GEMSOS directly. The structure definition is contained in Figure 13.
User_rec is a record of:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>Boolean</td>
<td>To determine if record is in use</td>
</tr>
<tr>
<td>Name</td>
<td>String[12]</td>
<td>User's login name</td>
</tr>
<tr>
<td>Pswd</td>
<td>String[12]</td>
<td>User's password</td>
</tr>
<tr>
<td>Max_class</td>
<td>Access Class</td>
<td>Maximum security class</td>
</tr>
<tr>
<td>Min_class</td>
<td>Access Class</td>
<td>Minimum security class</td>
</tr>
</tbody>
</table>

User_array is an array index 0 to Max_user of User_rec

Figure 13. Definition of the User Array

The structure of the message headers is given in Figure 14. Each user has two message header arrays, one for pending and the other for received message headers, for all of his messages, regardless of the classification level.

Messheading is a record of:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charclass</td>
<td>Character</td>
<td>Human readable security classification</td>
</tr>
<tr>
<td>Class</td>
<td>Access Class</td>
<td>GEMSOS readable security classification</td>
</tr>
<tr>
<td>From</td>
<td>String[8]</td>
<td>Message originator</td>
</tr>
<tr>
<td>Time</td>
<td>String[4]</td>
<td>Last edit time</td>
</tr>
<tr>
<td>Date</td>
<td>String[6]</td>
<td>Last edit date</td>
</tr>
</tbody>
</table>

Figure 14. Structure of the Message Header
This requires the read and write processes to be multilevel trusted processes. The design decision to use a multilevel header array was made early in the design process to ensure ease of use by the user. By having multilevel headers it is possible that a user could view a single screen containing the headers and find what message he had awaiting action regardless of the security class under which he is operating. The message headers are grouped into a record containing two arrays of nine elements, one array for incoming and one for outgoing messages, as shown in Figure 15. Each user's message header array is stored in a separate segment which is indexed by his user number.

The header array is an array 1..9 of messheading

Userhead is a record of:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Income</td>
<td>Theaderarray</td>
<td>Array of incoming messages</td>
</tr>
<tr>
<td>Outgo</td>
<td>Theaderarray</td>
<td>Array of outgoing messages</td>
</tr>
</tbody>
</table>

Figure 15. Structure of the Message Header Storage Structure

The final major data structure is the message itself as shown in Figure 16. The heading of the message is the same as that of the corresponding entry in the message heading array. The size of the body of the message was
Messtext is a record of:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heading</td>
<td>Messheading</td>
<td>Message header for the message</td>
</tr>
<tr>
<td>Body</td>
<td>Array of 2..23, 1..80 Character</td>
<td>Message text</td>
</tr>
</tbody>
</table>

Figure 16. Structure of a Message

determined by the requirement for a single screen editor. The array's index range corresponds to the line numbers on the screen display. This was done to facilitate the mapping of characters from the array to the screen. The messages are stored, by user, as segments with 18 messages of the same security classification per segment. The first nine messages are incoming messages and the remainder are the outgoing messages. This storage method creates at least 54 unused message spaces per user spread out over four segments. This method allows the different security classification to be stored as separate segments, allowing for single level segments with GEMSOS providing the security enforcement. Each of the messages can be uniquely identified by the security classification (which major branch), user number (which segment), and message number (location within the segment). The separation of security
levels and the ease of access offsets the wasted space in the storage of messages.

2. System Configuration Module

The system configuration module is the first process executed once the control of the machine has been passed to the application programs from the login process. The SSM can configure the Gemini TCB's terminal ports, within the security constraints stored in the system security memory. The maximum number of terminals that can be configured is determined at compile time of the System Configuration Module by a named constant embedded in the code. The maximum number of users must be known at sysgen time to construct the sufficient number of code and message segments. Once all of the desired terminals have been configured, the system configuration process then spawns all of the SMS node processes.

3. SMS User Control Menu Module

The SMS user control module acts as a master process for the individual users. It attaches the terminal to the process and then passes control to the user login module. When a user successfully gains access to the system he is then presented with a menu of options for him to select as shown in Figure 17. When the user selects a valid option, control is then passed to the appropriate module. Upon completion of an action, control is passed back to the SMS user control module and the menu is redisplayed. When the
SECURE MAIL SYSTEM

C. Create a message
E. Edit a message
R. Read message
S. Send a message
D. Delete a message
Q. Quit message editor

You have the following Messages:  

<table>
<thead>
<tr>
<th>Class</th>
<th>From</th>
<th>Time</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>U stewart</td>
<td>0721</td>
<td>880607</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Following messages are pending:

<table>
<thead>
<tr>
<th>Class</th>
<th>To</th>
<th>Time</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>U west</td>
<td>0715</td>
<td>880607</td>
</tr>
<tr>
<td>2</td>
<td>U stewart</td>
<td>0716</td>
<td>880607</td>
</tr>
<tr>
<td>3</td>
<td>U lengenfe</td>
<td>0717</td>
<td>880607</td>
</tr>
<tr>
<td>4</td>
<td>U adams</td>
<td>0718</td>
<td>880607</td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 17. SMS Main Menu

user exits the system, control is passed to the user login process.

4. User Login Module

This module is called by the user control module to verify the access authorization of the user. This process reads the user array segment to verify the login, password, and access level. Since this module reads a classified segment, it falls within the security perimeter and must be proved correct. After each login attempt the process detaches and then reattaches the terminal before passing control back to the calling procedure.
5. Create Message Module

This module creates a blank message form in memory after obtaining a message identification number from the header array if one is available. The system provides the security classification for the message (the current level at which the user is logged in), date and time of creation, and originator. The user provides the recipient's identifier. At this time the message is then passed to part of the edit module for the input of the text. After the user has completed editing the message he is given the option of saving the message or deleting it.

6. Read Message Module

This module allows the user to read a pending message, either incoming or outgoing without doing any modifications. One of the functions of this module is displaying a message on the screen as shown in Figure 18. The message display routine is used by the create and edit modules to display the message on the screen for further action.

7. Edit Message Module

This module allows the user to select any of the outgoing messages for editing. A subset of WordStar commands are used for the editing features. Appendix D contains the specific commands and their functions. None of the message header information can be changed by the user in this module. If the message was of lower security classification
Select one of the following Messages:

<table>
<thead>
<tr>
<th>Class</th>
<th>From</th>
<th>To</th>
<th>Time</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>U</td>
<td>pratt</td>
<td>west</td>
<td>0715</td>
</tr>
<tr>
<td>2.</td>
<td>U</td>
<td>pratt</td>
<td>stewart</td>
<td>0716</td>
</tr>
<tr>
<td>3.</td>
<td>U</td>
<td>pratt</td>
<td>lengenfe</td>
<td>0717</td>
</tr>
<tr>
<td>4.</td>
<td>U</td>
<td>pratt</td>
<td>adams</td>
<td>0718</td>
</tr>
<tr>
<td>5.</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.</td>
<td>No action</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 18. Message Selection Menu

than the current session, the security classification is changed to reflect the reclassification as a result of the modification. This prevents a user from circumventing the security classification system for the messages.

8. Send Message Module

The send module provides the user a means of transmitting a message to another user. The send message process is set up to allow the user to send a message that is classified at the current operating security level. The message header is displayed and the user is able to change the recipient of the message at this time. When the message is sent the system then updates the message header with the current data and time. A table look up is done on the user array to ensure that the recipient's name matches a user of the system. If no match is found the user is given the option of specifying a new name or aborting the message.
sending process. The system verifies that the recipient has access to the security classification of the message and an empty slot in the receive message header array. If the message is unable to be delivered that user is notified as such, without being given a reason. The message is not deleted from the message sender's message space by this module. This feature makes it possible to send a single message to multiple user without rekeying the message.

9. **Delete Message Module**

This module deletes the selected entry in the message header array and the message text. Both the message header array and the message text segments can be considered pooled resources. As such the data storage area must be overwritten by the delete process before the message can be considered deleted and the space reused. Since the delete option is a write operation, the user can only delete messages at the same security level where currently operating.

10. **Message Header Array Access Module**

This module is comprised of two major low level routines, the read and write header routines. These are trusted processes since the system maintains one header file for all of a user's messages regardless of the security level at which individual message were created. When a user enters the system at a classification lower then some of his pending messages, the header array will show the presence of
the messages by displaying the security classification of the message, but not the time, date, destination, or origin information that is contained in the header. This is done to allow the user to be alerted to the fact that he has the message but keeping the amount of information disclosed about the message to a minimum. This display of the security classification can be considered a covert storage channel in the system since the user can find out information concerning data of a higher security classification. Covert channels in the system will be discussed in a later section. This module is within the security perimeter, and has such the code has to be validated. The section on concurrency controls details how the read and write operations are accomplished.

11. Message Text Access Module

As in the message header access module, there are two major routines that comprise this module; read and write message operations. By placing a call to the read operation the user is able to read any of his messages that are at his current security level or a lower level. The write operation is strictly a single level operation. This is due to the use of the ticket primitive to ensure consistency of the data, as outlined in the chapter on eventcounts and sequencers. Both of the routines use the security features implemented in GEMSOS to ensure there are no unauthorized
data accesses. This module is within the security perimeter.

12. Terminal Control Module

This module is made up of two routines that are dependent on the type of terminal connected to the system. For this implementation, the DEC VT100 control set was chosen due to equipment availability. One of the routines provides the ability to clear the screen using the terminal control sequences. A direct cursor addressing procedure has been implemented. The direct cursor addressing is required by the message editor's cursor movement functions and the menus throughout the system. All of the procedures in this module write directly to the write device, rather than returning strings to the calling process.

F. CONCURRENCY CONTROL

Collectively the data segments can be thought of as a hierarchical database. The different security classes of message texts and the message headers form separate major branches. The message header branch then branches off in individual leaves for each user's message header. Each user has a leaf on each of the security branches if he has access to that security level. The leaves in turn are made up of the user's message texts for that security level. With this data base structure, the SMS is a database management process where multiple users are accessing the same data
base and their actions must be coordinated to ensure data consistency and verified to ensure the access is authorized.

The SMS will have a separate process running for each user node. Each process will be making updates to a multilevel security message database. This is the "multilevel secure readers-writers problem" that was presented in the previous chapter. To solve this problem, two modes of concurrency control are required: data access and process scheduling.

1. **Data Access Control**

   In the "multilevel secure readers-writers problem" we have the constraints that a process can read lower level data, write higher level data, and modify data at the same level. GEMSOS provides the required security checks on the eventcounts and sequencers as an integral part of the data segments. This eliminates the need for any explicit checking of access authorization for the security classes in the code.

   The use of eventcounts and sequencers is limited to the synchronizing of access to data. Read operations, which can be done by more than one process concurrently with no loss of consistency in the system, use only eventcounts to ensure that the data is in a consistent state. By using only eventcounts it is possible to read any data from the same or lower security classes. The eventcount is read at two points in the system read process, once before the data
is read and once after. If the two values of the eventcount are the same, there were no writes during the read process. If the values are different the read aborts and restarts. This ensures that the data is read in a consistent state. Figure 19 contains the pseudocode for the algorithm.

```
Make security branch mentor known
Make message segment known
Swapin message segment
Make pointer to the segment
REPEAT
  Read Segment's Eventcount
  Move selected data to desired data structure
  Read segment's eventcount
UNTIL the two eventcount values are equal
Terminate message segment
Terminate security branch mentor
```

Figure 19. Sample Read Operation

The writing process must provide a means for the writer to gain exclusive control (only one write operation at a time) of the data to ensure consistency of the data. To do this, the ticket operation is used; it is the only mechanism that provides mutual exclusion. The single level security limitation of the ticket operation prohibits writing data at any security level except the current security level. After every write operation, the appropriate eventcount is advanced to ensure the data is in
a consistent state. A pseudocode implementation of the write process can be found in Figure 20.

![Pseudocode](image)

**Figure 20. Sample Write Operation**

2. **Process Scheduling**

   As outlined in Chapter II, GEMSOS is capable of multiprogramming and multiprocessing. Due to the structure of the SMS, with one master process creating the node processes, all processes are run on a single processor, the multiprocessing feature is not used by the SMS. This makes use of the multiprogramming scheduling algorithm in GEMSOS. The "run to block" algorithm is used by the system. Each of the processes will run until a request is placed for a service that can not be immediately provided, at which time
it will block and a ready process will be allowed to commence execution.

G. COVERT CHANNELS

As with most secure systems, covert channels exist in the SMS. In this section the channels found during a search of the system and code will be discussed, rationalized and an estimated bandwidth given. A channel naming system that was presented in [Gass88] is used to identify the type of channel. The author has not had formal training in the evaluation of secure systems and as such more, covert channels may exist and the computed bandwidth may be incorrect. The bandwidths computed in this section tend to be overly pessimistic in that it would be impossible for a user to sustain the channels at the computed bandwidths.

1. Storage Channels

The message header array is an object attribute channel. When the array is displayed on the terminal it is possible to find out the security class of all the user's messages. Given that there are a maximum of nine messages and four possible security classes for each message the most information that can be leaked is 36 bits. Assuming that no more then one screen display per second is possible, the maximum band width is 32 bits per second. The actual bandwidth will be considerably smaller since much of the header array will remain static for a length of time. This covert channel, while it has a large potential bandwidth, is
not deemed serious. The justification for this is that a user is authorized access to the data that is being leaked to him by the channel.

The message header also presents the opportunity to use an object existence channel. By sending a user repeated messages it is possible to compute the number of messages that were pending prior to the attack. From the total of nine messages, four bits are required to identify explicitly the number of messages. In the worst case one attempt is required to determine that the recipient has nine messages. This action will take about one second for a bandwidth of four bits per second. This channel is created by the static nature of the header array. A variable length header array, such as a linked list, would not have this channel.

It is possible for a user to create an object existence channel to determine the maximum security class of each user on the system. This can be done by attempting to send highly classified mail to a known user downgrading each successive message until a message is received. With the four security classes, four bits of data are leaked out in each attempt. These four bits times the number of users equals the maximum amount of information that can be gained by this channel. Assuming all users are at the highest classification level tried, at best case it would take four to five seconds per attempt. The resulting bandwidth would be about one bit per second. The justification for allowing
the existence of this channel is that most users know each
others security clearances in advance since such information
is readily available.

2. **Timing Channels**

Knowing that the Gemini Computer uses a run to block
scheduling algorithm, it is possible for a knowledgeable
user to make a rough determination of the system load by the
delay in the services provided by the system. Write access
to the data segments is controlled by the ticket mechanism,
which allows one user at a time to access the data. This
delay would make it possible for a user to determine if
other users were trying to access the same data segment.
These two timing channels can be defeated by installing a
random length delay loop in the sections of code that reads
and writes the data to the segments. The channels' exis-
rence can be rationalized by the fact that the
perpetrator cannot compute the correct number of users on
the system—just a rough idea of that number.
V. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

It was the purpose of this thesis to demonstrate that it is possible to design a parallel multilevel secure process that is simple for a user to operate. The result was a menu driven electronic mail system that allows up to 12 users and four levels of security classification. This system demonstrates that the goal of the thesis is attainable.

Programming in a secure and parallel environment requires a different "mind set" than conventional unsecure single process programming. For a secure environment the programmers must know the classification of the data and which sections of code can access particular data. In a parallel environment the system's designers must impose controls on data access to ensure all reads and writes are atomic (no other process can alter the data during the transaction) and no data is left in an inconsistent state.

Security can be built into a system with minimum overhead and additional expense. This is true if the system designers consider security as an integral part of the system.

Eventcounts and sequencers are an efficient and simple way to control access to shared data by parallel processes.
The fact that reading and writing of eventcounts can be separated make them ideal for use in a secure environment.

B. RECOMMENDATIONS

The author severely underestimated the skill level and expertise needed to program in a secure environment. For this reason, if the Gemini computers owned by the Naval Postgraduate School are to be used, an ongoing research program must be developed. This will allow experience to be passed from one thesis student to the next in a series of follow on theses.

The Gemini TCB is capable of supporting up to 48 terminals, whereas, SMS currently limits the number of terminals and users to 12. The data structures of the system can be modified to allow the SMS to support more than 12 users.

The message storage formats used by the SMS waste considerable amount of space. A more efficient means of storage would be to create a new segment for every message under a mentor segment unique to each security class and user pair.

The use of secure computers warrants considerable study. A secure computer is a special purpose computer, and as such should be used for specific applications. Programming a secure computer is considerably more difficult than a normal computer. The increased difficulty is offset by the benefits of a secure system for certain applications. If
the system has an overriding security requirement then a secure computer should be considered.

The Naval Postgraduate School has the resources available to develop a comprehensive program to explore the uses of secure computers. Such a program should include the development of applications for, and the management of, secure computers. This program should be under a larger and more general Automated Data Processing (ADP) security program.

C. FINAL COMMENT

Computers can be made only as secure as the least trusted individual who has access to them. To paraphrase a common cliche, "Computers don't leak information, people do."
APPENDIX A

USER MANUAL FOR THE SECURE MAIL SYSTEM

A. INTRODUCTION

The Secure Mail System (SMS) is a multilevel secure electronic mail system. It can support up to 12 users on 12 active terminals. Four separate security classifications of messages are used by the system to segregate the messages.

The system has been divided into two logical areas. System initialization is done by the System Security Manager (SSM) at boot time. This process then spawns the node process in which all user interaction takes place.

B. SYSTEM INITIALIZATION

During the system initialization process the SSM configures the system by specifying the active terminals and the security classification for the selected terminals. A menu is presented to select the terminal from a list of terminals that are connected to the system. Once the SSM selects a terminal he is then prompted for a security class. After the node parameters have been selected, the configuration process spawns the node process. This is repeated until all terminals have been configured. At this time the configuration process blocks.
C. NODE PROCESS

The node process runs on each terminal and provides the user interaction. The user is required to verify his identity by of a username/password login sequence. At this point the session security class is selected. If the user does not provide a correct login sequence within three tries, the terminal process blocks. The SSM must then restart the system.

Upon successful login into the system the user is presented with a menu of options and a listing of incoming and outgoing messages. The options include edit, create, read, delete, send and quit. The after selecting the edit option the user is then prompted to make a selection from a list of messages. Once the menu has been selected the user is presented with the text that had been previously entered into the message. From here any of the commands listed in Table A-1 can be used. When the user exits the editing mode the date, time and the security classification are updated to reflect the current system parameters.

By selecting the create option the user indicates that he desires to create a new message form. This can only be done when there is an empty space in the message header array. The user is prompted for the destination of the message, the rest of the header is filled in by the system. The system then goes into the edit mode to allow the user to fill in the text of the message. After completion of
TABLE A-1
EDIT MODE COMMANDS

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>^e</td>
<td>Cursor up</td>
</tr>
<tr>
<td>^x</td>
<td>Cursor down</td>
</tr>
<tr>
<td>^s</td>
<td>Cursor left</td>
</tr>
<tr>
<td>Bksp</td>
<td>Cursor left</td>
</tr>
<tr>
<td>^d</td>
<td>Cursor right</td>
</tr>
<tr>
<td>^m</td>
<td>Go to first position of the next line</td>
</tr>
<tr>
<td>Return</td>
<td>Go to first position of the next line</td>
</tr>
<tr>
<td>^a</td>
<td>Go to start of current line</td>
</tr>
<tr>
<td>^f</td>
<td>Go to end of current line</td>
</tr>
<tr>
<td>^r</td>
<td>Go to top of message text</td>
</tr>
<tr>
<td>^c</td>
<td>Go to bottom of message text</td>
</tr>
<tr>
<td>^i</td>
<td>Tab, move cursor 5 spaces right</td>
</tr>
<tr>
<td>^v</td>
<td>Turn insert mode on, any control character turn insert mode off</td>
</tr>
<tr>
<td>^g</td>
<td>Delete a character</td>
</tr>
<tr>
<td>^y</td>
<td>Delete a line</td>
</tr>
<tr>
<td>^z</td>
<td>Exit edit mode</td>
</tr>
</tbody>
</table>

editing the user is asked if they would like to save the message and for the destination of the message.

The read option allows the user to view a message without the ability to edit it. This is useful when he desires to consult a lower level security message.

Upon selecting the delete option the user is queried to find out if he desires to delete an incoming or outgoing message. Once that selection has been made a list of the messages is displayed. From this menu the user selects the
message number for deletion. Both the header and the message text are overwritten in this process.

When the user desires to send a message he is presented with a list of messages available for transmission. When he selects one of the messages the security class is checked and upgraded if it is lower than the current session security level. The user is prompted for the destination of the message. The data and time are updated before the message is sent. This process does not delete the message, thereby allowing the same message to be sent to multiple users without rekeying.

The quit option logs the user out and allows a new user to login to the system at a new security class.

D. SUMMARY

The SMS was designed to be a simple user friendly secure electronic mail system. Commercially available unsecure electronic mail systems offer many more features but do not offer the security that is built into this system.
APPENDIX B

SMS CONFIGURATION MODULE CODE

The source code for this module contains routines that are proprietary to Gemini Computers. In order to allow unlimited distribution, the code has not been included in this thesis. Code for this module is available from the WAR lab custodian at the following address:

Superintendent, Code 55wg
Naval Postgraduate School
Monterey, CA 93943
APPENDIX C

SMDESS CODE

1 (This code was written in Pascal MT+ version 3.0. It is
2 linked using the following command line for the Linkmt
3 program:
4
5 Linkmt sms = f:rl-init,sms,f:lf30/s,rllib/s,paslib/s/p:80
6
7 The code is shared among all node processes, each node has a
8 separate stack and data segment.)
9 ($e-)
10 ($K0) ($K1) ($K2) ($K3) ($K5) ($K6) ($K7)
11 ($K8) ($K9) ($K10) ($K12) ($K13)
12
13 module sms;
14
15 const
16   {i f:gate-con.zli}
17   {i f:rl-con.zli}
18   {i f:user-con.zli}
19 (*   {i f:cd-con.zli} *)
20 (program specific constants)
21   userseg = 11;
22   headerseg = 2;
23   useg = 3;
24   cseg = 4;
25   sseg = 5;
26   tsseg = 6;
27 type
28   {i f:gate-typ.zli}
29   {i f:lib-typ.zli}
30   {i f:kst-typ.zli}
31   {i f:rlp-typ.zli}
32   {i f:user-typ.zli}
33 (program specific types)
34   {i sms.typ}
35
36 (* External Declarations *)
37
38 {i f:lib.zli}
39 {i f:io-str.zli}
40 {i f:gate.zli}
41 {i f:loadregs.zli}
42 {i include.dec}
43
This is the node process

```pascal
procedure main(var init: r1_process_def);

var
success,result : integer;
user : string;
minclass,maxclass,userclass : access_class;
choice : char;
innotout : boolean;
tryctr : integer;
portno : integer;

procedure sendmess(user:string8;userclass:access_class);

var
messnum,result,destno,sendnum : integer;
choice,yesno : char;
headerarray : theaderarray;
message : messtext;

begin
clrscr;
Selectfile (user,userclass,FALSE,choice);
if choice <> '0' then
begin
messnum := ord(choice) - 48;
readheader(user,FALSE,userclass,headerarray,result);
readfile(FALSE,userclass,messnum,headerarray[messnum],
message,result);
disp2(message);
repeat
    putstr(w_dev,'Is this the correct message? (Y/N) ');
    getchar(r_dev,yesno);
    putln(w_dev,' ');
    until yesno in ['y','n','Y','N'];
if yesno in ['y','Y'] then
begin
    repeat
        putstr(w_dev,'What is the new destination? ');
        getchar(r_dev,yesno);
        putln(w_dev,' ');
        until yesno in ['y','n','Y','N'];
    if yesno in ['N','n'] then
begin
    ```
getln(r_dev, message. heading. reci);
end;
getusernum(message. heading. reci, sendnum, result);
if result = 0 then
begin
findheaderslot(message. heading. reci, userclass,
TRUE, destno);
if destno <> 0 then
begin
(send the message)
writeheader(message. heading. reci, TRUE, userclass,
message. heading, destno, result);
writefile(TRUE, userclass, destno, message. heading,
message, result);
(update user's version)
writeheader(user, FALSE, userclass, message. heading,
messnum, result);
writefile(FALSE, userclass, messnum, message. heading,
message, result);
putln(w_dev, 'The message has been sent');
end
else
begin
putln(w_dev, 'Unable to deliver the message');
putstr(w_dev, 'Check destination's username');
putln(w_dev, ' and security class');
end
end;
else
begin
putln(w_dev, 'Unable to deliver the message');
putstr(w_dev, 'Check destination's username');
putln(w_dev, ' and security class');
end;
end;
end;

PROCEDURE hookup console(portno: integer; wrt_dev: integer;
rd_dev: integer);
var
success: integer;
begin
repeat
attach(portno, wrt_dev, false, success);
until (success = no_error);
repeat
attach(portno, rd_dev, true, success);
until (success = no_error);
PROCEDURE userlogin(VAR user: string8; Var userclass: access_class; maxclass: access_class; portno: integer; Var result: integer);

var
    password : string8;
    userin : char;
    usernum : integer;
begin
    result := 0;
    userin := 'U';
    userclass := maxclass;
    hookuup_console(portno, w_dev, r_dev);
    clrscr;
    putstr(w_dev, 'Login: ');
    getln(r_dev, user);
    putstr(w_dev, 'Password (will not echo): ');
    noecho getln(r_dev, password);
    putln(w_dev, '1');
    repeat
        putstr(w_dev, 'Desired access class (U/C/S/T): ');
        getchar(r_dev, userin);
        putln(w_dev, '1');
        until userin in ['U', 'C', 'S', 'T', 'u', 'c', 's', 't'];
    if ord(userin) > 96 then
        userin := chr(ord(userin) + 32);
    case userin of
        'U': userclass.compromise[0] := unclass_level;
        'C': userclass.compromise[0] := conf_level;
        'S': userclass.compromise[0] := secret_level;
        'T': userclass.compromise[0] := t_secret_level;
    end;
    ( detach(w_dev);
    detach(r_dev); )
(* do look up for username, password, access class *)
lookupuser(user, password, userclass, usernum, result);
if (userclass.compromise[0] > maxclass.compromise[0])
    and (result = 0) then result := 4;
( hookuup_console(portno, w_dev, r_dev); )
end;

(*)
PROCEDURE lookupuser(user, password: string8; userclass: access_class;
var usernum: integer; var result: integer);
var
    arrayptr : userptr;
    seg_number, size, cntr : integer;
    mclass : access_class;
begin
(*putln(w_dev,'in the look up user proc');*)
mclass := init.resources.max_class;
seg_makeknown(init.initial_seg[2], Userseg, seg_number, 
r_w, size, mclass, result);
swapin_segment(seg_number, result);
arrayptr := lib_mk_pntr(lidt_table, seg_number, 1);
cntr := 0;
while (cntr < max_user) and 
not (arrayptr^[cntr].name = user) do
begin
  cntr := cntr + 1;
end;
if arrayptr^[cntr].name = user then
begin
  usernum := cntr;
  if password = arrayptr^[usernum].pswd then
begin
  if (userclass.compromise[0] <=
      arrayptr^[usernum].max_class.compromise[0]) then
    result := 0;
  end
else
  result := 2;
end;
else
  result := 1;
seg_terminate(seg_number, cntr);
end;

PROCEDURE getusernum(user:string8; Var usernum, result: integer);
var
arrayptr : userptr;
seg_number, cntr, size : integer;
mclass : access_class;
begim
mclass := init.resources.max_class;
seg_makeknown(init.initial_seg[2], Userseg, seg_number, 
r_w, size, mclass, result);
show_err('get usernum makeknown result = ', result);
swapin_segment(seg_number, result);
show_err('get usernum swapin result = ', result);
arrayptr := lib_mk_pntr(lidt_table, seg_number, 1);
cntr := 0;
while (cntr < max_user) and (arrayptr^[cntr].name <> user) do
begin
  cntr := cntr + 1;
end;
if (userclass.compromise[0] <=
    arrayptr^[cntr].max_class.compromise[0])
and (arrayptr^[cntr].name = user) then
begin
result := 0;
usernum := cntr;
end
else
result := 1;
seg terminate(seg_number, cntr);
show err('set usernum set terminate result = ',cntr);
end;

(*)
PROCEDURE writefile(innotout:boolean;sec class:access_class;
messnum:integer;mhead:messheading;
Var message:messtext;VAR result:integer);

var
usernum : integer;
size,seg1,seg2,evc1,evc2,cntr : integer;
arrayptr : messptr;
user :string8;
branch : integer;
tampptr : varpointer;
mespptr : ^messtext;
begin
{ensure no write down is allowed}
if sec_class.compromise[0] >= mhead.class.compromise[0] then
begin
if innotout then
user := mhead.recip
else
user := mhead.from;
case mhead.class.compromise[0] of
  unclass level : branch := useg;
  conf_level : branch := cseg;
  secret_level : branch := sseg;
  t_secret_level : branch := tsseg;
  else result := 2;
end; {case}
getusernum(user,usernum,result);
seg_makeknown(init.initial_seg[2],branch,seg1,
r_w,size,sec_class,result);
show_err('write file make known result 1 = ',result);
seg_makeknown(seg1,usernum,seg2,
r_w,size,sec_class,result);
show_err('write file make known result 2 = ',result);
swapin_segment(seg2,result);
show_err('write file swapin result = ',result);
ticket(seg2,evcl,result);
show_err('write file ticket result =',result);
await(seg2,evcl,result);
show_err('write file await result =',result);
temp.ptrseg := lib_mk_sel(ldt_table,seg2,1);
if innotout then
  temp.ptr.offset := (messnum - 1) * sizeof(messtext);
else
  temp.ptr.offset := (messnum + 8) * sizeof(messtext);
  messptr := temp.ptr.p;
  messptr^ := message;
advance(seg2,result);
show_err('write file advance result =',result);
seg_terminate(seg2,result);
seg_terminate(seg1,result);
end
else
  result := 1;
end;
(*}
PROCEDURE readfile(innotout:boolean;sec_class:access_class;
messnum:integer;mhead:messheading;
Var message:messtext;VAR result:integer);
var
  usernum : integer;
  size,seg1,seg2,evcl,evc2,cntr : integer;
  arrayptr : messptr;
  user:string8;
  branch : integer;
  mclass : access_class;
begin
  result := 0;
  (ensure no read up is allowed)
  if sec_class.compromise[0] >= mhead.class.compromise[0] then
    begin
      if innotout then
        user := mhead.recip
      else
        user := mhead.fro;
      case mhead.class.compromise[0] of
        unclass_level : branch := useg;
        conf_level : branch := cseg;
        secret_level : branch := sseg;
        t_secret_level : branch := tsseg;
else result := 2;
end; (case)
if result = 0 then
begin
    getusernum(user, usernum, result);
    mclass := init.resources.max_class;
    seg_makeknown(init.initial_seg[2], branch, seg1,
       r, size, mclass, result);
    show_err('read file make known result 1 = ', result);
    seg_makeknown(seg1, usernum, seg2,
       r, size, mclass, result);
    show_err('read file make known result 2 = ', result);
    swapin_segment(seg2, result);
    show_err('read file swapin result = ', result);
    repeat
        read_evcs(seg2, evc1, result);
        arrayptr := lib_mk_ptr(ldt_table, seg2, l);
        if innotout then
            message := arrayptr[messnum]
        else
            message := arrayptr[messnum + 9];
        read_evcs(seg2, evc2, result);
        until evc1 = evc2;
        seg_terminate(seg2, result);
    seg_terminate(seg1, result);
end;
else result := 1;
end;
(*)
PROCEDURE readheader(user: string8; innotout: boolean;
access: access_class; VAR headerarray: theaderarray;
VAR result: integer);
var
seg1, seg2, evc1, evc2, cntr, size : integer;
arrayptr : headptr;
mclass : access_class;
usernum : integer;
begin
(*println(w dev,'accessing the header array list');*)
getusernum(user, usernum, result);
mclass := init.resources.max_class;
seg_makeknown(init.initial_seg[2], headerseg, seg1, 
    r_w, size, mclass, result);
show_err('read header make known result 1 = ', result);
seg_makeknown(seg1, usernum, seg2, 
    r_w, size, mclass, result);
show_err('read header make known result 2 = ', result);
swapin_segment(seg2, result);
show_err('read header swapin result = ', result);
repeat
    read_evc(seg2, evc1, result);
    arrayptr := lib_mk_ptr(ldt_table, seg2, 1);
    if innotout then
        headerarray := arrayptr^.income
    else
        headerarray := arrayptr^.outgo;
    read_evc(seg2, evc2, result);
until evc1 = evc2;
seg_terminate(seg2, result);
seg_terminate(seg1, result);
for cntr := 1 to 9 do
    if access.compromise[0] <
        headerarray[cntr].class.compromise[0] then
        with headerarray[cntr] do
            begin
                from := ' '; 
                reci := ' '; 
                time := ' '; 
                date := ' ';
            end;
end;
PROCEDURE writeheader(user: string8; innotout: boolean;
    access: access_class; header: messheading; messnum: integer;
    Var result: integer);
var
    size, usernum : integer;
    seg1, seg2, cntr, evc1 : integer;
    arrayptr : headptr;
    mclass : access_class;
begin
(*putln(w_dev, 'accessing the header array list');*)
getusernum(user, usernum, result);
mclass := init.resources.max_class;
seg_makeknown(init.initial_seg[2], headerseg, seg1, 
    r_w, size, mclass, result);
show_err('write header make known result 1 = ', result);
seg_makeknown(seg1, usernum, seg2,
456     r_w, size, mclass, result);
457     show_err('write header make known result 2 = ', result);
458
459     swapin_segment(seg2, result);
460     show_err('write header swapin result = ', result);
461
462     ticket(seg2, evcl, result);
463     show_err('write header ticket result = ', result);
464
465     await(seg2, evcl, result);
466     show_err('write header await result = ', result);
467
468     arrayptr := lib_mk_ptr(ldt_table, seg2, 1);
469     header.class := access;
470     if innotout then
471         arrayptr.income[messnum] := header
472     else
473         arrayptr.outgo[messnum] := header;
474     advance(seg2, result);
475     show_err('write header advance result = ', result);
476
477     seg_terminate(seg2, result);
478
479     seg_terminate(segl, result);
480
481 END;
482
483 PROCEDURE get_return;
484
485 var
486     tempstr: string;
487
488 begin
489     putstr(w_dev, '<ret> to continue');
490     getln(r_dev, tempstr)
491 end;
492
493 PROCEDURE write_comp(class: access_class; var str: string);
494
495 begin
496     case class.compromise[0] of
497         unclass_level: str := 'Unclassified';
498         conf_level: str := 'Confidential';
499         secret_level: str := 'Secret';
500         t_secret_level: str := 'Top Secret';
501     end;
502 end;
503
504 PROCEDURE gotoxy(col, row: integer);
505 var
vstr : string;
strlen : integer;

begin
if ((0 < col) and (col <= 80) and
(1 <= row) and (row <= 24)) then
begin
vstr[1] := chr(27);
vstr[2] := '1';
strlen := 3;
if row > 9 then
begin
vstr[strlen] := chr(48 + (row div 10));
strlen := 4;
end;
vstr[strlen] := chr(48 + (row mod 10));
strlen := strlen + 1;
vstr[strlen] := '1';
strlen := strlen + 1;
if col > 9 then
begin
vstr[strlen] := chr(48 + (col div 10));
strlen := strlen + 1;
end;
vstr[strlen] := chr(48 + (col mod 10));
strlen := strlen + 1;
vstr[strlen] := 'H';
vstr[0] := chr(strlen);
end
else
begin
vstr[0] := chr(1);
vstr[1] := chr(7);
end;
putstr(w_dev,vstr);
end;

PROCEDURE clrscr;
var
vstr : string;
ctrr : integer;
begin
for cttrr := 1 to 25 do
putln(w_dev,'');

vstr[0] := chr(4);
vstr[1] := chr(27);
vstr[2] := '1';
putstr(w_dev,vstr)
PROCEDURE mainscreen(sec_class:access_class;
user:string8;var choice:char);

( main menu screen )

var

headerarray : theaderarray;
innotout : boolean;
class_str: string;
cntr, result : integer;

begin

innotout := TRUE;
clrscr;
putstr(w_dev,' Security class: ');
write comp( sec_class,class_str);
putln(w_dev,class_str);
putstr(w_dev,' ');
putln(w_dev,'SECURE MAIL SYSTEM ');
pztl(w_dev,
putln(w_dev,' C. Create a message ');
putln(w_dev,'
putln(w_dev,' D. Delete a message ');
putstr(wdev,' Q. Quit message editor ');
putln(w_dev,'
putln(wdev,'
putln(w_dev,'i. 1. ');
putln(wdev,'2. ');
putln(wdev,'3.1. ');
putln(wdev,'4. ');
putln(wdev,'5. ');
putln(wdev,'6. ');
putln(wdev,'7. ');
putln(wdev,'8. ');
putln(wdev,'9. ');

{incoming messages}
innotout := true;
Readheader(user,innotout,sec_class,headerarray,result);

for cntr := 1 to 9 do
begin
goxy(5,cntr + 15);
putchar(w_dev,headerarray[cntr].charclass);
goxy(8,cntr + 15);
putstr(w_dev,headerarray[cntr].from);
goxy(18,cntr + 15);
putstr(w_dev,headerarray[cntr].time);
goxy(25,cntr + 15);
putstr(w_dev,headerarray[cntr].date);
end;

innotout := false;
Readheader(user,innotout,sec_class,headerarray,result);

for cntr := 1 to 9 do
begin
goxy(46,cntr + 15);
putchar(w_dev,headerarray[cntr].charclass);
goxy(51,cntr + 15);
putstr(w_dev,headerarray[cntr].reci);
goxy(62,cntr + 15);
putstr(w_dev,headerarray[cntr].time);
goxy(70,cntr + 15);
putstr(w_dev,headerarray[cntr].date);
end;

repeat
goxy(28,10);
getchar(r_dev,choice);
if ord(choice) > 96 then
  choice := chr(ord(choice) - 32);
  until (choice = 'C') or (choice = 'E') or
  (choice = 'R') or (choice = 'S') or (choice = 'D')
  or (choice = 'Q');
end;

(*)
PROCEDURE Selectfile (user:string8;sec_class:access_class;
innotout:boolean;VAR choice:char);
(from here the user selects on of the available of files)

var
  headerarray : theaderarray;
  cntr,result : integer;
  fileflag : boolean;
begin
662 clrscr;
663 putln(w_dev, ' ');  
664 putln(w_dev, ' ');  
665 putln(w_dev, ' ');  
666 putstr(w_dev,' Select one of the ');  
667 putln(w_dev,' following Messages:');  
668 putln(w_dev,' Class From To');  
669 putln(w_dev,' Time Date');  
670 putln(w_dev,' 1.');  
671 putln(w_dev,' 2.');  
672 putln(w_dev,' 3.');  
673 putln(w_dev,' 4.');  
674 putln(w_dev,' 5.');  
675 putln(w_dev,' 6.');  
676 putln(w_dev,' 7.');  
677 putln(w_dev,' 8.');  
678 putln(w_dev,' 9.');  
679 putln(w_dev,' 0. No action');  
680 
681 (read in the header file)  
682 Readheader(user, innotcut, sec_class, headerarray, result);  
683 
684 for cntr := 1 to 9 do  
685 begin  
686   gotoxy(19,cntr + 5);  
687   putchar(w_dev,headerarray[cntr].charclass);  
688   gotoxy(27,cntr + 5);  
689   putstr(w_dev,headerarray[cntr].from);  
690   gotoxy(38,cntr + 5);  
691   putstr(w_dev,headerarray[cntr].reci);  
692   gotoxy(48,cntr + 5);  
693   putstr(w_dev,headerarray[cntr].tm);  
694   gotoxy(59,cntr + 5);  
695   putstr(w_dev,headerarray[cntr].date);  
696 end;  
697 gotoxy(10,16);  
698 (make sure the user selects a valid file or no action)  
699 repeat  
700   getchar(r_dev,choice);  
701   if (choice >= '1') and (choice <= '9') then  
702     fileflag :=  
703     headerarray[ord(choice) - 48].charclass <> '*';  
704   if choice = '0' then  
705     fileflag := TRUE;  
706   until (choice >= '0') and (choice <= '9') and Fileflag;  
707 clrscr;  
708 end;  
709  
710  
711 (*)  
712 PROCEDURE int2str(num:byte;var str:string);  
713
begin
str[1] := '0';
if num >= 10 then
str[1] := chr(48 + (num div 10));
str[2] := chr(48 + (num mod 10));
str[0] := '2';
end;

(*)
PROCEDURE gettime(var time:string4;var date:string6);
const
clockslot = 5;
var
str :string;
result : integer;
clockbuff : cd_tin_buff;
begin
  cd_r_attach(clockslot,result);
  cd_r_dev(clockslot,clockbuff,result);
  int2str(clockbuff[2],str);
  time[0] := chr(4);
  time[1] := str[1];
  int2str(clockbuff[3],str);
  time[3] := str[1];
  int2str(clockbuff[7],str);
  date[0] := chr(6);
  date[1] := str[1];
  int2str(clockbuff[5],str);
  date[3] := str[1];
  int2str(clockbuff[6],str);
  date[5] := str[1];
detach(clockslot);
end;

(*)
PROCEDURE deleteheader(user:string;sec_class:Access_class;
 availslot:integer;innotout:boolean;Var result:integer);
var
header : messheading;
begin
  header.charclass := '*';
  header.class.compromise[0] := 0;
  header.from := ' ';
  header.reci := ' ';
  header.time := ' ';
  header.date := ' ';
  writeheader(user,innotout,sec_class,header,availslot,result);
PROCEDURE delmessage(user: string8; sec_class: access_class;
choice: integer; innotout: boolean;
result: integer);

var
message: messtext;

cntrl, cntr2: integer;

begin
with message.heading do
begin

    charclass := 'x';
    class.compromise[0] := 0;
    if innotout then
        begin
            reci := user;
            from := ' ';
        end
    else
        begin
            from := user;
            reci := ' ';
        end;

time := ' ';
date := ' ';

end;

for cntrl := 2 to 23 do
    for cntr2 := 1 to 80 do
        message.body[cntrl, cntr2] := ' ';

writefile(innotout, sec_class, choice, message.heading,
    message, result);

end;

PROCEDURE deletemess(user: string8; sec_class: access_class);

var
innotout: boolean;
choice, yesno: char;
headerarray: theaderarray;
messnum, result: integer;
message: messtext;

begin
clrscr;

repeat
    putstr(w_dev, 'Do you want to delete an ');
    putstr(w_dev, 'message? (Y/N) ');
    getchar(r_dev, yesno);
    putln(w_dev, ' ');
    until yesno in ['y', 'n', 'Y', 'N'];
innotout := (yesno = 'Y') or (yesno = 'y');
Selectfile (user, sec_class, innotout, choice);
if choice <> '0' then
  begin
    messnum := ord(choice) - 48;
    readheader (user, innotout, sec_class, headerarray, result);
    readfile (innotout, sec_class, messnum, headerarray [messnum],
               message, result);
    disp2 (message);
    repeat
      putstr (w_dev, 'Is this the correct message? (Y/N) ');
      getchar (r_dev, yesno);
      println (w_dev, ' ');
    until yesno in ['y', 'n', 'Y', 'N']
    if yesno in ['y', 'Y'] then
      begin
        deleteheader (user, sec_class, messnum, innotout, result);
        delmessage (user, sec_class, messnum, innotout, result);
      end;
  end;
end;
end;

{ PROCEDURE findheadslot (user: string8; sec_class: access_class;
  innotout: boolean;
  Var availslot: integer);

  var
    cntr : integer;
    headerarray: theaderarray;
    result : integer;

  begin
    readheader (user, innotout, sec_class, headerarray, result);
    availslot := 1;
    while (availslot < 9) and
      (headerarray[availslot].charclass <> '*') do
      availslot := availslot + 1;
    if headerarray[availslot].charclass <> '*' then
      availslot := 0
    else
      begin
        with headerarray[availslot] do
          begin
            class := sec_class;
            case class.compromise[0] of
              unclass_level : charclass := 'U';
              conf_level : charclass := 'C';
              secret_level : charclass := 'S';
              t_secret_level : charclass := 'T';
              else charclass := '*';
            end;
            from := user;
            gettime (time, date)
```plaintext
870   end;
871   writeheader(user,innotout,sec_class,
872                 headerarray[availslot],availslot,result);
873   end;
874 end;
875
876 (*)
877 PROCEDURE createmess(user : string8;sec_class:access_class);
878 var
879   cntrl1,cntr2,availslot : integer;
880   innotout : boolean;
881   message : messtext;
882   yesno : char;
883   result : integer;
884
885 begin
886   clrscr;
887   innotout := FALSE;
888   findheadslot(user,sec_class,innotout,availslot);
889   if availslot = 0 then
890     begin
891       putstr(w_dev,'There is no room in the header array for ');
892       writeln(w_dev,'any more messages');
893       putstr(w_dev,'You must delete and/or send some
894                  of them before');
895       writeln(w_dev,' you can create any more');
896     end
897   else
898     begin
899       updateheader(user,sec_class,message. heading,result);
900       for cntrl1 := 2 to 23 do
901         for cntr2 := 1 to 80 do
902           message.body[cntrl1,cntr2] := ' ';
903           dispmessage(message);
904           edit(message);
905           clrscr;
906           repeat
907             putstr(w_dev,'Do you want to save the message? (Y/N)' );
908             getchar(r_dev,yesno);
909             writeln(w_dev,' ');
910           until yesno in ['Y','N','y','n'];
911           if (yesno = 'Y') or (yesno = 'y') then
912             begin
913               updateheader(user,sec_class,message. heading,result);
914               writeheader(user,innotout,sec_class, message. heading,
915                             availslot,result);
916               writefile(innotout,sec_class,availslot,
917                           message. heading,message,result);
918             end
919           else
920             deleteheader(user,sec_class,availslot,innotout,result);
```
921    end;
922 end;
923
924 (*
925 PROCEDURE disp(message: messtext);
926 var
927    cntrl, cntr2: integer;
928    secstring : string;
929    message : messtext;
930
931 begin
932    message := message;
933
934    clrscr;
935    write_comp(message, heading, class, secstring);
936    putstr(w_dev, secstring);
937    gotoxy(25,1);
938    putstr(w_dev, message, heading, from);
939    gotoxy(35,1);
940    putstr(w_dev, message, heading, reci);
941    gotoxy(45,1);
942    putstr(w_dev, message, heading, time);
943    gotoxy(55,1);
944    putln(w_dev, message, heading, date);
945    gotoxy(1,2);
946    for cntrl := 2 to 23 do
947        begin
948          for cntr2 := 1 to 79 do
949              putchar(w_dev, message, body[cntrl, cntr2]);
950              putln(w_dev, message, body[cntrl, 80]);
951          end;
952          gotoxy(60,24);
953          write_comp(message, heading, class, secstring);
954        end;
955
956 (*
957 PROCEDURE disp2(message: messtext);
958 var
959    secstring : string;
960 begin
961    write_comp(message, heading, class, secstring);
962    putstr(w_dev, secstring);
963    gotoxy(25,1);
964    putstr(w_dev, message, heading, from);
965    gotoxy(35,1);
966    putstr(w_dev, message, heading, reci);
967    gotoxy(45,1);
968    putstr(w_dev, message, heading, time);
969    gotoxy(55,1);
970    putln(w_dev, message, heading, date);
971    gotoxy(1,2);
PROCEDURE readmess(user:string8;sec_class:access_class);
const
  innotout = TRUE;
var
  choice : char;
  headerarray : theaderarray;
  messnum,result : integer;
  message : messtext;
begin
  Selectfile (user,sec_class,innotout,choice);
  if choice <> '0' then
    begin
      messnum := ord(choice) - 48;
      readheader(user,innotout,sec_class,headerarray,result);
      readfile(innotout,sec_class,messnum,headerarray[messnum],
               message,result);
      if result <> 0 then
        begin
          clrscr;
          writeln(w_dev,'Error reading the message');
          get_return;
        end
      else
        begin
          dispmessage(message);
          get_return;
        end;
    end;
  end;

PROCEDURE updateheader(user:string8;sec_class:access_class;
var
  headerrec:messheading;var result:integer);
var
  tempstr : string;
  cntr : integer;
  clockbuff : cd_tim_buff;
begin
  with headerrec do
    begin
      class := sec_class;
      case class.compromise[0] of
        unclass level : charclass := 'U';
        conf level : charclass := 'C';
        secret level : charclass := 'S';
        t_secret level : charclass := 'T';
        else charclass := '*';
      end;
      from := user;
gettime(time, date);
putstr(w_dev, 'Name of person to send message to? ');
getln(r_dev, reci);
end;
end;

(*)

PROCEDURE edit(var message : messtext);
var
  inchar : char;
  cntr, cntr2, charnum, colcntr, linecntr : integer;
begin
  gotoxy(1,2);
  linecntr := 2;
  colcntr := 1;
  repeat
    getchar(r_dev, inchar);
    charnum := ord(inchar);
    if charnum in [32..126] then
      begin
        if (linecntr = 23) and (colcntr = 80) then
          putchar(w_dev, chr(7))
        else
          if (colcntr = 80) then
            begin
              putchar(w_dev, inchar);
              message.body[linecntr, colcntr] := inchar;
              linecntr := linecntr + 1;
              colcntr := 1;
              gotoxy(colcntr, linecntr);
            end
          else
            begin
              pitchar(w_dev, inchar);
              message.body[linecntr, colcntr] := inchar;
              colcntr := colcntr + 1;
              end;
        end
      else
        case charnum of
          (Cursor Movement) (up "^E")
            5 : begin
              if linecntr = 2 then
                putchar(w_dev, chr(7))
              else
                begin
                  linecntr := linecntr - 1;
                  gotoxy(colcntr, linecntr);
                end;
            end;
1078 {down "^X"}
1079 24 : begin
1080 if linecntr = 23 then
1081     putchar(w_dev, chr(7))
1082 else
1083     begin
1084         linecntr := linecntr + 1;
1085         gotoxy(colcntr, linecntr);
1086     end;
1087 end;
1088 {left "^S"}
1089 19,8 : begin
1090 if (linecntr = 2) and (colcntr = 1) then
1091     putchar(w_dev, chr(7))
1092 else
1093     if (colcntr = 1) then
1094         begin
1095             linecntr := linecntr - 1;
1096             colcntr := 80;
1097             gotoxy(colcntr, linecntr);
1098         end
1099 else
1100         begin
1101             colcntr := colcntr - 1;
1102             gotoxy(colcntr, linecntr);
1103         end;
1104 end;
1105 {right "^D")
1106 4 : begin
1107 if (linecntr = 23) and (colcntr = 80) then
1108     putchar(w_dev, chr(7))
1109 else
1110     if (colcntr = 80) then
1111         begin
1112             linecntr := linecntr + 1;
1113             colcntr := 1;
1114             gotoxy(colcntr, linecntr);
1115         end
1116 else
1117         begin
1118             colcntr := colcntr + 1;
1119             gotoxy(colcntr, linecntr);
1120         end;
1121 (Carrage return "^M", or the "return" key)
1122 13 : begin
1123 if linecntr = 23 then
1124     putchar(w_dev, chr(7))
1125 else
1126     begin
1127         linecntr := linecntr + 1;
1128         }
1129    colcntr := 1;
1130    gotoxy(colcntr,linecntr);
1131    end;
1132  end;
1133  (Start of line "^A")
1134    1 : begin
1135        colcntr := 1;
1136        gotoxy(colcntr,linecntr);
1137    end;
1138  (End of line "^F")
1139    6 : begin
1140        colcntr := 80;
1141        gotoxy(colcntr,linecntr);
1142    end;
1143  (Top of page "^R")
1144    18 : begin
1145        linecntr := 2;
1146        gotoxy(colcntr,linecntr);
1147    end;
1148  (Bottom of Page "^C")
1149    3 : begin
1150        linecntr := 23;
1151        gotoxy(colcntr,linecntr);
1152    end;
1153  (Tab, five spaces "^I", or the "tab" key)
1154    9 : begin
1155        if colcntr < 75 then
1156            begin
1157                colcntr := colcntr + 5;
1158                gotoxy(colcntr,linecntr);
1159            end;
1160    end;
1161
1162  (insert "^V")
1163    22 : begin
1164        gotoxy(70,1);
1165        putstr(w_dev,'INSERT ON');
1166        gotoxy(colcntr,linecntr);
1167        getchar(r_dev,inchar);
1168        charnum := ord(inchar);
1169        while charnum in [32..126] do
1170            begin
1171                if (colcntr < 80) then
1172                    begin
1173                        for cntr := 9 downto colcntr do
1174                            message.body[linecntr,cntr + 1] :=
1175                                message.body[linecntr,cntr];
1176                        message.body[linecntr,colcntr] := inchar;
1177                        for cntr := colcntr to 80 do
1178                            putchar(w_dev,
1179                                message.body[linecntr,cntr]);
1180                end;
1181        end;
colcntr := colcntr + 1;
gotoxy(colcntr,linecntr);
end
else
begin
putchar(w_dev,inchar);
message.body[linecntr,colcntr] := inchar;
inchar := chr(27); (exit)
end;
getchar(r_dev,inchar);
charnum := ord(inchar);
end;
gotoxy(70,1);
putstr(w_dev,'');
gotoxy(colcntr,linecntr);
end;

(Delete single character "^G")
7 : begin
for cntr := colcntr to 79 do
begin
message.body[linecntr,cntr] :=
message.body[linecntr,cntr + 1];
putchar(w_dev,message.body[linecntr,cntr]);
end;
message.body[linecntr,80] := ' ';
putstr(w_dev,' ');
gotoxy(colcntr,linecntr);
end;

(Delete Line "^Y")
25 : begin
colcntr := 1;
gotoxy(colcntr,linecntr);
for cntr := linecntr to 22 do
begin
message.body[cntr] := message.body[cntr+1];
for cntr2 := 1 to 80 do
putchar(w_dev,message.body[cntr,cntr2]);
end;
for cntr2 := 1 to 80 do
begin
message.body[23,cntr2] := ' ';
putstr(w_dev,' ');
end;
gotoxy(colcntr,linecntr);
end;

(Exit edit mode "^2")
26 : {exit};
(Invalid key error, sound bell)
else
putchar(w_dev,chr(7));

end;

t until charnum = 26;
end;

(*)

PROCEDURE editmessage(user: string8; sec_class: access_class);

const
  innotout = FALSE;

var
  choice, yesno : char;
  headerarray : theaderarray;
  messnum, result : integer;
  message : messtext;

begin
  Selectfile (user, sec_class, innotout, choice);
  if choice <> '0' then
    begin
      messnum := ord(choice) - 48;
      readheader(user, innotout, sec_class, headerarry, result);
      readfile(innotout, sec_class, messnum,
               headerarray[messnum], message, result);
      if result <> 0 then
        begin
          clrscr;
          pitln(w_dev,'Error reading the message');
          get return;
        end
      end else
        begin
          dispmessage(message);
          edit(message);
          clrscr;
          repeat
            putstr(w_dev,'Do you want to save the changes? (Y/N)');
            getchar(r_dev,yesno);
            until yesno in ['Y','N','y','n'];
            if (yesno = 'Y') or (yesno = 'y') then
              begin
                updateheader(user, sec_class,
                             headerarray[messnum], result);
                writeheader(user, innotout, sec_class,
                             headerarray[messnum],
                             messnum, result);
                writefile(innotout, sec_class, messnum,
                           headerarray[messnum], message, result);
              end;
        end;
end;
end;
end;
end;
PROCEDURE show_err(str: string; code: integer);
begin (show_err)
if code <> no_error then
begin
putstr(w_dev,str);
putstr(w_dev,' ');
pitdec(w_dev,code);
putln(w_dev,' ');
end;
end; (show_err)

(* the port number is passed in from the SMSMAIN program*)
portno := init.reserved[0];
minclass := init.resources.min_class;
maxclass := init.resources.max_class;
trycntr := 0;
while trycntr < 3 do
begin
userclass := maxclass;
clrscr;
userlogin(user,userclass,maxclass,portno,result);
if result = 0 then
begin
trycntr := 0;
choice := ' ';
innotout := True;
while choice <> 'Q' do
begin
mainscreen(userclass,user,choice);
case choice of
'C' : begin
createmess(user,userclass);
end;
'R' : begin
readmess(user,userclass);
end;
'E' : begin
editmessage(user,userclass);
end;
'S' : begin
sendmess(user,userclass);
end;
'D' : begin
deletemess(user,userclass);
end;
'Q' : begin
(Quit branch of case statement, No operation)
clrscr;
detach(w_dev);
detach(r_dev);
end;
end;
end;
end;

end;
end;
end;
end;
else
begin
    show err('bombed out error = ',result);
detach(w_dev);
detach(r_dev);
    trycntr := trycntr + 1;
end;
end;
clrscr;
putln(w_dev,'This terminal has been locked out due to too ');
putln(w_dev,'many wrong login/password attempts');
putln(w_dev,'Notify the system manager for reactivation');
putln(w_dev,'Termination of Secure Mail System');
detach( w_dev );
detach( r_dev );
self delete(init.initial_seg[0], success);
repeat until (false);
end;
end;
end.
modend.
APPENDIX D

SMS TYPE INCLUDE FILE

This file contains the type declarations that are covered in Chapter IV.

```
string4 = string[4];
string6 = string[6];
string8 = string[8];

messheading = record
  charclass : char;
  class : access class;
  from : string8;
  reci : string8;
  time : string4;
  date : string6;
end;

messtext = record
  heading : messheading;
  body : array [2..23,1..80] of char;
end;

theaderarray = array[1..9] of messheading;
messarray = array[1..9] of messtext;

userhead = record
  income : theaderarray;
  outgo : theaderarray;
end;

usermess = array [1..18] of messtext;

userptr = ^user_array;
messptr = ^usermess;
headptr = ^userhead;
```
LIST OF REFERENCES


108


## INITIAL DISTRIBUTION LIST

<table>
<thead>
<tr>
<th>No.</th>
<th>Distribution List</th>
</tr>
</thead>
</table>
| 1.  | Defense Technical Information Center  
      Cameron Station  
      Alexandria, Virginia  22304-6145 | 2 |
| 2.  | Library, Code 0142  
      Naval Postgraduate School  
      Monterey, California  93943-5002 | 2 |
| 3.  | Director, Information Systems (OP-945)  
      Office of the Chief of Naval Operations  
      Navy Department  
      Washington, D.C.  20350-2000 | 1 |
| 4.  | Commandant of the Marine Corps (Code TE06)  
      Headquarters, United States marine Corps  
      Washington, D.C.  20360-0001 | 1 |
| 5.  | Computer Technology Programs, Code 37  
      Naval Postgraduate School  
      Monterey, California  93943-5000 | 1 |
| 6.  | CDR Joseph S. Stewart, Code 55St  
      Naval Postgraduate School  
      Monterey, California  93943-5000 | 2 |
| 7.  | Major Richard A. Adams, Code 52Ad  
      Naval Postgraduate School  
      Monterey, California  93943-5000 | 1 |
| 8.  | 1st LT David R. Pratt  
      SABARS/MCCDPA  
      Code G9S, Bldg 3041A  
      MCCDC,  
      Quantico, Virginia  22134-5001 | 3 |
| 9.  | Chairman, Computer Science Department, Code 52  
      Naval Postgraduate School  
      Monterey, California  93943-5000 | 1 |