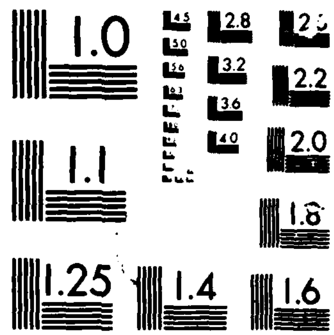
AD-A175 908 DYNAMIC SHARING OF THE SYSTEM RESOURCES IN MULTILEVEL SECURE SYSTEM(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA M A REYES 26 SEP 86 1/2

UNCLASSIFIED F/G 9/2 NL


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

25-1450

(2)

AD-A175 908

NAVAL POSTGRADUATE SCHOOL

Monterey, California

DTIC ELECTE JAN 1 4 1987 S D

D

THESIS

DYNAMIC SHARING OF THE SYSTEM RESOURCES IN MULTILEVEL SECURE SYSTEM

by

Miguel Angel Reyes

September 1986

Thesis Advisor: Gary S. Baker

Approved for public release; distribution is unlimited

DTIC FILE COPY

87 1 13 043

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION: unclassified	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Naval Postgraduate School	6b OFFICE SYMBOL (If applicable)
7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School	
6c ADDRESS (City, State, and ZIP Code): Monterey, California 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, California 93943-5000
8a NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
DYNAMIC SHARING OF THE SYSTEM RESOURCES IN MULTILEVEL SECURE SYSTEM

12 PERSONAL AUTHOR(S) Reyes, Miguel Angel

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM ______ TO ______	86 September 26	125

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	sharing system resources; Gemini Trusted Multiple Microcomputer Base machine; Janus/Ada

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This research represents a preliminary step in the development of a reliable application program simulating an operating system which handles several multi-security-level users dynamically sharing system resources in the Gemini Trusted Multiple Microcomputer Base machine.

The proposed design presents the necessary steps to follow when working in a multilevel configuration. The use of primitives that support the application design are explained along with a description of the implementation of this application using Janus/Ada language. In addition, security constraints are identified and system test results are described.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION: unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL: CDR Gary S. Baker	22b TELEPHONE (Include Area Code): (408)646-2073 — 22c OFFICE SYMBOL: 52Bj

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

Approved for public release, distribution is unlimited

Dynamic Sharing of The System Resources In Multilevel Secure System

by

Miguel A. Reyes
Major, Peruvian Air Force
B. S., Peruvian Air Force Academy, 1973

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

September 1986

Author: Miguel A. Reyes

Approved by: Gary S. Baker, Thesis Advisor

Uno R. Kodres

Uno R. Kodres, Second Reader

Vincent Y. Lum, Chairman,
Department of Computer Science

Kneale T. Marshall

Kneale T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

This ~~research~~ represents a preliminary step in the development of a reliable application program simulating an operating system which handles several multi-security-level users dynamically sharing system resources in the Gemini Trusted Multiple Microcomputer Base machine.

The proposed design presents the necessary steps to follow when working in a multilevel configuration. The use of primitives that support the application design are explained along with a description of the implementation of this application using Janus/Ada language. In addition, security constraints are identified and system test results are described. Keywords: computer programming; systems engineering.

DISCLAIMER

The reader is cautioned that computer programs developed in the research may not have been validated. However every effort has been made to ensure that the programs are free of computational and logical errors. The nature of this research and the time available were not sufficient to validate completely all the software developed.

Many terms used in this thesis are registered trademarks of commercial products. Rather than attempt to cite each individual occurrence of a trademark, all registered trademarks appearing in this thesis will be listed below :

1) Gemini Computers Inc., Monterey California
   - Gemini Trusted Multiple Computer Base
   - GEMSOS
2) RR Software Inc., Madison, Wisconsin
   - Janus/Ada Development Package
3) Digital Research, Pacific Grove, California
   - Pascal MT+
   - CP/M-86
4) Intel Corporation, Santa Clara, California
   - INTEL
   - MULTIBUS
   - APX-286
5) Xerox Corporation, Stamford, Conneticut
   - ETHERNET, Local Area Network
6) Digital Equipment Corporation, Maynard, Massachusetts
   - Vax 11/780 Minicomputer
   - Unix Operating System

TABLE OF CONTENTS

I. INTRODUCTION 9
A. GENERAL DISCUSSION 9
B. THESIS FORMAT 11
II. BACKGROUND 13
A. TRUSTED COMPUTER SYSTEM 13
B. GEMINI TRUSTED MULTIPLE MICROCOMPUTER BASE 15
1. General Information 15
2. Resources Management 16
a. Segment Management 17
b. Process Management 17
c. Device Management 17
3. Gemini Security Architecture 18
4. Naval Postgraduate Scholl Version of Gemini 19
III. SOFTWARE DEVELOPMENT OVERVIEW 20
A. GENERAL DESCRIPTION 20
B. HIERARCHICAL STORAGE STRUCTURE 21
C. I/O DEVICE ASSIGNMENT 24
D. PROCESS CREATION 24
1. Create / Makeknown Segments Module 26
2. Address Space Specification Module 27
3. Register Record Module 28
4. Resource Record Module 28
5. Create Process Module 29
E. LOCAL DESCRIPTION TABLE (LDT) 29
F. CREATE/DELETE SEGMENT 30
G. MAKEKNOWN/TERMINATE SEGMENT 30
H. SWAPIN/SWAPOUT SEGMENT 31
I. SYNCHRONIZATION 31
IV. SYSTEM DESIGN 32
A. INITIAL DESIGN 32
1. Objective 32
2. Initial Design Constraints 32
B. COMPROMISE DESIGN 34
1. Objective 34

2. Actual Design Constraints 34
3. Hierarchical Structure Design 36
C. SYSTEM SOFTWARE DESIGN 39
1. Application Programs Design 39
a. Load Parameters Module 40
b. Create Segments Module 40
c. Create Process Module 42
d. Synchronization Module 42
e. Delete Processes Module 44
f. Terminate Segments Module 44
g. Delete Segments Module 45
2. User Handler Application Program 45
a. Load Parameter module 45
b. Create Segments Module 45
c. Create Process Module 46
d. Synchronization Module 46
3. Active User Application Program 48
a. Attach_terminal Module 48
b. Loop Module 48
c. Detach_terminal Module 49
d. Self_delete Module 49
V. IMPLEMENTATION 50
A. GENERAL DISCUSSION 50
1. Primitive Calls 50
2. Auxiliary Functions 51
B. IMPLEMENTATION CONSTRAINTS 53
C. IMPLEMENTATION STEPS 54
1. Single User, Single Security Access Level 54
2. Several Users, Sigle Security Access Level 55
3. Several Users, Multilevel Security Access 55
D. SYSGEN SUBMIT FILE 57
VI. CONCLUSIONS AND RECOMMENDATIONS 58
A. CONCLUSIONS 58
1. System Operation 58
2. System Performance 59
B. RECOMMENDATIONS 59
1. Hardware Improvements 59

2. Software Improvements 59
3. Future Research 60
a. Directly Related to this Research 60
b. Related to Secure Mass Storage System 60
APPENDIX A - MAIN APPLICATION PROGRAM (CCP) 61
APPENDIX B - USER HANDLER APPLICATION PROGRAM 73
APPENDIX C - ACTIVE USER APPLICATION PROGRAM 81
APPENDIX D - PRBDOS APPLICATION PROGRAM 86
APPENDIX E - COMMON PROCEDURES UTILITY 90
APPENDIX F - SERVICE ROUTINES AND ADDITIONAL DATA STRUCTURE 105
APPENDIX G - SYSGEN SUBMIT FILE (SSB) 122
LIST OF REFERENCES 123
INITIAL DISTRIBUTION LIST 124

LIST OF FIGURES

1.1 Microcomputer System Organization 11
3.1 Hierarchical Structure 22
3.2 Hierarchical Structure including an application 23
3.3 Process Creation Main Modules 27
4.1 Initial Design 33
4.2 Actual Design 35
4.3 System : Hierarchical Structure 37
4.4 Address Space Specification for each user process 38
4.5 Hierarchical Structure : User Processes 38
4.6 Main Application Program 40
4.7 System Parameters 41
4.8 User Active Application Program Modules 47
5.1 Main Application Program LDT 56

I. INTRODUCTION

A. GENERAL DISCUSSION

This thesis presents the design and part of the implementation of software for the Gemini Computer, under CP/M-86 and GEMSOS Operating Systems, to allow the dynamic sharing of system resources in a multilevel secure computer system, using Janus/Ada language as a host language.

Security has traditionally been provided by physical measures (fences, police, dogs, alarms, etc.) to prevent unauthorized access to computers. But today this is no longer sufficient. The extensive use of networks brings the possibility of uncontrolled access to the resources of any installation from remote sites. Discretionary security measures, i.e., "password" types, alone are not totally adequate where security of information is paramount [Ref. 1]. Steps must be taken to strictly reinforce a non-discretionary policy as well. This situation provides enough motivation to look for more reliable methods to control and limit the access.

This necessity of Trusted Computers is even more critical and compulsory in military organizations, where many delicate decisions depend on the quality of the information, which if known or modifiable by unauthorized users, creates a risk to the country's security.

The Naval Postgraduate School is involved in the use of microcomputers in its Microcomputer Laboratory in which several of them are networked together through a concentrator for information sharing. These systems are to be used by people with different levels of security clearance who handle information with multiple security classifications. This application necessitates the use of a mass storage system with the ability to limit access to classified programs and data. The only effective way to insure multi-level internal security is by employing a **Trusted Computer Base** [Ref. 2] such as provided by the Gemini computer.

Figure 1.1 shows the proposed configuration of a microcomputer system having the Gemini computer at its base. The Gemini System provides the base layer of an operating system which implements internal information security through a **"security kernel"** design [Ref. 3:pp. 1-2]. To construct the architecture proposed in Figure 1.1 requires implementing the top layer of the operating system for handling the Input/Output operations. Three design elements can be identified :

1) The Concentrator. The concentrator will contain a software "crossbar switch" which allows dynamic switching for I/O interconnection between attached systems.

2) The Dynamic Assignment of Security Access Levels to I/O devices. In this aspect. the main idea is to manage the access level of the terminal without relating it to an specific connection. The access level should be dynamically recognized by the characteristics of the user. rather than be limited to a secondary issues such as location or terminal number. This is the main topic addressed by the current research.

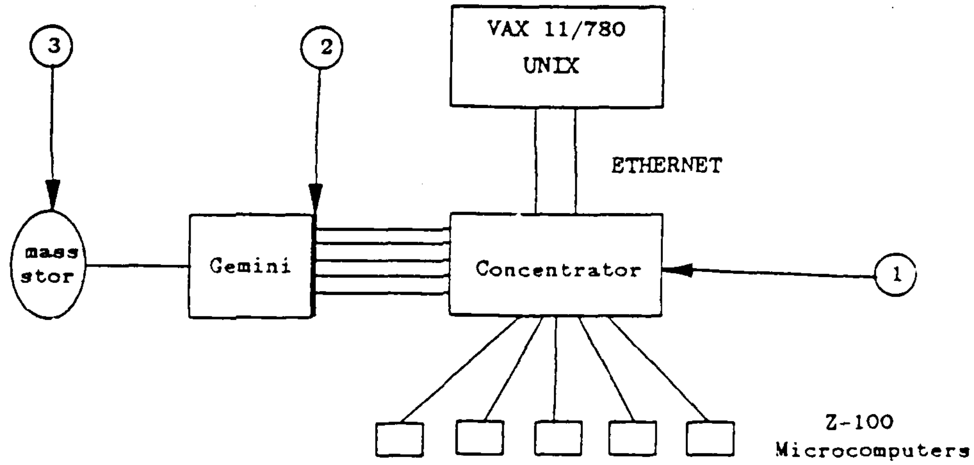


Figure 1.1 Microcomputer System Organization

3) A Segmented "File" System for Mass Storage. The purpose of this system would be to provide a one-level segmented storage system for mass secondary storage (hard disk) within a secure environment.

B. THESIS FORMAT

This thesis is composed of six chapters organized in such a way as to provide the reader with the background necessary to understand internal multilevel computer security concepts, in particular dynamic sharing of system resources. Simple guidelines in software development are introduced and a design is presented for the implementation of a prototype system which allows several users with different clearance levels to share system resources.

Chapter I provides general information focusing on reasons why Multilevel Secure Systems are important.

Chapter II addresses the background necessary to understand Multilevel Security concepts and explains the Gemini System Architecture and possibilities.

Chapter III provides specific information related to the steps necessary to produce basic application programs in the Gemini System.

Chapter IV describes the design of a small "Operating System" application program that will handle dynamic sharing of resources, in particular I/O.

Chapter V presents the description of the support modules used to develop application programs, and describes the implementation constraints and the steps performed to complete an application.

Chapter VI discusses the results obtained from this research effort. It also suggests future investigations in this field as a continuation of the work performed in this thesis.

II. BACKGROUND

A. TRUSTED COMPUTER SYSTEM

Most of the basic concepts and definitions mentioned in this section are referenced to the standardization performed by the Department of Defense (DoD) related to Trusted Computer Systems. These standards are contained in "DoD Trusted Computer System Evaluation Criteria" [Ref. 2] published in 1983: it lists definitions and concepts, and provides detailed criteria pertaining to the test and evaluation of trusted computers.

The security policies considered are mandatory (non-discretionary) security and discretionary security.

Mandatory Security is defined as :

> Security Policies defined for systems that are used to process classified or other specifically categorized sensitive information must include provisions for the enforcement of mandatory access control rules. That is, they must include a set of rules for controlling access based directly on a comparison of the individual's clearance or authorization for the information and the classification or sensitivity designation of physical and other environmental factors of control. The mandatory access control rules must accurately reflect the laws, regulations, and general policies from which they are derived. [Ref. 2:p. 72]

Discretionary Security is defined as :

> Security Policies that are defined for systems that are used to process classified or other sensitive information must include provisions for the enforcement of the discretionary access control rules. That is, they must include a consistent set of rules for controlling and limiting access based on

identified individuals who have been determined to have a Need-to-Know for the information. [Ref. 2:p. 73]

As the names imply. mandatory policy contains a set of rules ·.at are imposed on all users in the organization. and discretionary policy is a specific set of rules further limiting access on a "need-to-known" basis.

A multilevel secure system in a conventional computer system is impossible to attain without a way to enforce the policies previously indicated. Security can be broken without knowledge of the user because intentionally or not, there are possible unsecure points that will allow access to the system. A typical example of this problem is a "Trojan Horse" program [Ref. 4:p. 66] provided by a third source which may have code intentionally "hidden" with the purpose of copying the user's access control code [Ref. 5:pp. 55-56] when the user executes the program. This represents an illegal condition.

The mandatory (non-discretionary) and discretionary policies are implemented in a "security kernel" which provides mechanisms for limiting the access to the information. Security Kernel is defined as the hardware and software that realizes the "reference monitor" abstraction (i.e.. the realization of these limiting policies), and in turn provides the idea of protection in which the active entities (subjects). such as people or computer programs. make reference to passive entities (objects). such as documents or segments of memory. using a set of current access authorizations [Ref. 6:p. 14]. The access class is divided into [Ref. 6:p. 16] :

a) Compromise (observe) which states that a subject can not "observe" the contents of an object unless the access class of the subject is greater than or equal to the access class of the object.

b) Integrity (modify) which stipulates that a subject may not "modify" an object unless the object's access class is greater than or equal to the access class of the subject.

The multilevel secure system considered in this research is focused on the access of many users to common system resources without restriction of a designated resource to a specific kind of user. Specifically, any user can utilize any terminal, and the access class in not limited to physical terminals with fixed access levels. The user priviledges. will be determined during a logon process when the user provides his username and password.

Additional explanations and details concerning secure communication methods and possible threats involved are described in [Ref. 7:pp. 15-28] and [Ref. 8:pp. 19-21].

B. GEMINI TRUSTED MULTIPLE MICROCOMPUTER BASE

1. General Information

Gemini Trusted Multiple Microcomputer Base was the computer used in the research of this thesis. It represents an advance in technology combining several concepts, Multilevel Security, Multiprocessing and Multiprogramming, to provide an important Trusted Base Machine that can be considered for a wide range of computer applications where security is a fundamental consideration. Actually, it has been evaluated by the U.S. Department of Defense for certification to meet the B3 class [Ref. 3:pp. 1-2]. and is currently undergoing evaluation in a specialized application for A1 class.

The major features of this system are [Ref. 3] :

1) Up to 8 Intel iAPX286-Base microcomputers are connected on the same Multibus.

2) Minimization of bus contention by locating data and code segments in the local memory of each microcomputer, whenever possible.

3) Capability of multiprocessing and multiprogramming. The Gemini Secure Operating System (GEMSOS) can multiplex many virtual processors onto a single physical processor. and support combinations of parallel and pipelined processing.

4) Connection of different storage and I/O devices using an RS-232 Interface Board which can handle up to 8 ports.

5) The system includes a Bus Controller. a Real-Time Clock. a Data Encryption Device (NBD-DES. Algorithm). a Non-volatile Memory for storing system passwords and encryption keys. It also provides a System-Unique Identifier.

6) Each iAPX286 microprocessor supports 4 hierarchical privilege levels.

7) CP/M-86 Operating System is used for software development and several different programming languages are available to develop application software.

8) Modular Expansion and Configuration Independence.

2. Resources Management

At the heart of the Gemini computer system is the GEMSOS operating system as previously mentioned.

GEMSOS resource management services are invoked by an application program through a set of calls to a collection of subroutines which represent an interface between GEMSOS and the application. Each language compiler has a unique interface library [Ref. 3:p. 4]. GEMSOS manages three classes of entities : segments, processes, and devices.

a. Segment Management

All the information is stored in logical objects called segments which are handled by the application programmer using segment management calls. These operating system calls are described in detail in [Ref. 9:p. 10].

b. Process Management

A process can be viewed as an application program that runs under the control of GEMSOS to perform some specific activity. The process is created by the application program using service calls related to the process management described in [Ref. 3:pp. 5-8] and [Ref. 9:p. 23]. In addition to Process Management, there are additional concepts related to process synchronization in which the application programmer has tools to sequence processes that communicate with each other. Synchronization is obtained utilizing eventcounts and sequencers [Ref. 10:pp. 115-124] associated with processes. A working explanation will be provided in Chapter IV. Process Synchronization calls are described in detail in [Ref. 3:pp. 6-7] and [Ref. 9:p. 33].

c. Device Management

The design of the I/O management functions of the Kernel are novel. The basic idea consists of reducing the code needed in the Kernel to control I/O functions, by incorporating many of the details within the application program.

The result is a reduction of the Kernel's size and the verification is easier. Security checks are performed only when the device is attached to a process [Ref. 3:pp. 8-9]. I/O management service calls are described in [Ref. 11:p. 38].

3. Gemini Security Architecture

Since the iAPX286 Microcomputer supports 4 protection levels. GEMSOS uses these levels to enforce the security critical layering of the system. Protection levels are called Ring 0 thru Ring 3. with Ring 0 the most privileged ring [Ref. 3: p.10]. Ring 0 supports the Mandatory (non-discretionary) Policy and Ring 1 contains the Discretionary Policy. the combination of which comprising the Security Perimeter and the greater portion of GEMSOS.

Ring 1 also holds functions such as user authentication. system security officer functions and audit functions. Ring 2 is used for common services utilized by many users. e.g.. Database Management System: Ring 3 is used as application layers for programs and data: both are outside of the Security Perimeter and can be used during the system development process (i.e.. as in developing the upper layer of an application system as is the focus of this research). and for the execution environment for user's programs.

Process management in the GEMSOS architecture is through the use of a two level traffic controller design (inner traffic controller or bottom level. and upper traffic controller).

The inner traffic controller binds a physical processor with a fixed number of "virtual processors". Two of these are used to support system services (an idle virtual processor and a manager virtual processor) and the others are available to the upper level traffic controller. The inner traffic controller also provides the primitives for synchronization between virtual processors emulating a multiprogramming configuration.

The upper level traffic controller multiplexes a number of processes onto the virtual processors defined by the inner traffic controller. These functions are performed in each of the physical processors comprising the Gemini computer (up to 8) [Ref. 7:pp. 14-15], through a distributed operating system.

4. Naval Postgraduate School Version of Gemini

a. One APX286 Microcomputer
b. Two 1.2 Mbyte floppy disk drives
c. One RS-232 Interface Board (max 8 ports)

With this configuration, GEMSOS must multiplex the processes created onto virtual processors and then onto a single physical processor. The synchronization primitives support the communication among processes. It is important to note that in this configuration, a multiprocessing environment does not exist. The potential for exploiting processor parallelism does not exist.

III. SOFTWARE DEVELOPMENT OVERVIEW

A. GENERAL DESCRIPTION

This chapter provides the foundation for the necessary steps to develop software in Gemini machines. It is important because it provides the basic components that are needed. Considering that Gemini is a new concept in Multilevel Secure machines, it is still not user friendly. A bridge between the application programmer and the operating system service primitives had to be created in order to develop reliable software.

The basic Gemini operating system is limited and only supports those operating system functions which are concerned with system security. Thus, only an operating system base is provided upon which an upper level i.e., I/O handler, file manager, etc. must be provided to support specific user requirements.

Subroutines or modules were prepared to perform the interface between the programmer and the operating system primitives. A complete explanation of these modules is provided in the design and implementation chapters.

The objective of this chapter is to present an ordered method of developing application software within the environment. It should be considered as a guide and not as a fixed set of rules. The facts considered here are taken from the user's manual Ref. 9

B. HIERARCHICAL STORAGE STRUCTURE

The Gemini System provides a one-level secondary storage system for information (In the NPS configuration. secondary storage refers to floppy disks). File concepts are not supported by Gemini. but instead segments are used which are considered as objects having logical attributes related to them (i.e.. security) and being of a maximum 64 K bytes size. Segmentation is extended to secondary storage, providing the one-level storage system.

The segments are handled by the system as a hierarchy. where each segment is identified by a unique path name. This segment's "handle" corresponds to the index of an entry in the Local Description Table (LDT) of the process that creates and/or uses the segment. As such. a single segment can have many different handles depending on where and how many processes enter it into their respective LDT tables. But it has only one unique path name in the hierarchy.

The representation of segments follow a hierarchical structure in which the root is the System Root (transparent to the user) and the whole collection of segments is assembled as an inverted tree. Each user's program is part of the hierarchical structure and it is declared as a segment that is statically created at system generation time using the Sysgen Submit file explained in [Ref. 11:pp. 12-14].

System generation consists of creating a hierarchical structure of all segments known to the system at system runtime. in particular at system "Bootload". It basically is the inclusion of the segment hierarchy comprising the upper levels of the application target system, into the provided base level hierarchy that is known as GEMSOS. This is accomplished by utilizing segments declared in the submit file, i.e., the sysgening process creates a hierarchy using the segments indicated in the submit file.

Figure 3.1 shows a typical hierarchical structure representing the entries that GEMSOS requires to run programs. This structure is fixed and must be considered in the development of any application program. Figure 3.2 shows the addition of segments necessary to execute a specific application. Entry 5 in the hierarchical structure is always dedicated as the "root" of user applications. This entry is the root or mentor of all the segments that are needed to implement the upper levels of the system application program. Under the Gemini concept, a segment can support up to 12 descendants (entries) numbered from 0 thru 11. To

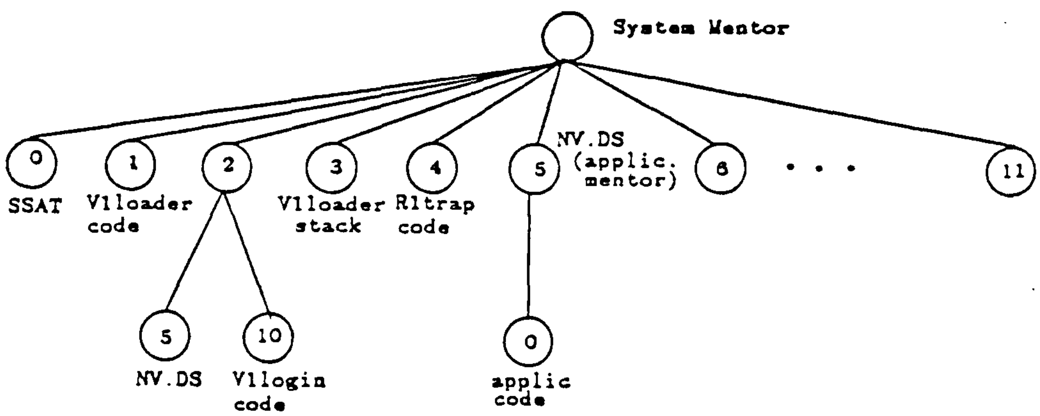


Figure 3.1 Hierarchical Structure

support this concept there is an "aliasing" table that relates the segments to their mentors; a segment can only have one mentor [Ref. 11:p. 1].

When an entry is used, it cannot be assigned again until a delete segment call marks this segment as available. The numbers indicated in both figures correspond to entry numbers of segments associated with a mentor (from 0 thru 11); segment numbers have a different enumeration, which correspond to their entries in the LDT.

Each segment in the system has a unique pathname that identifies it, but this identifier is not used by the application processes. Instead a process-local segment number is used. When two processes share the same segment, each one recognizes the segment by the number assigned in its own LDT.

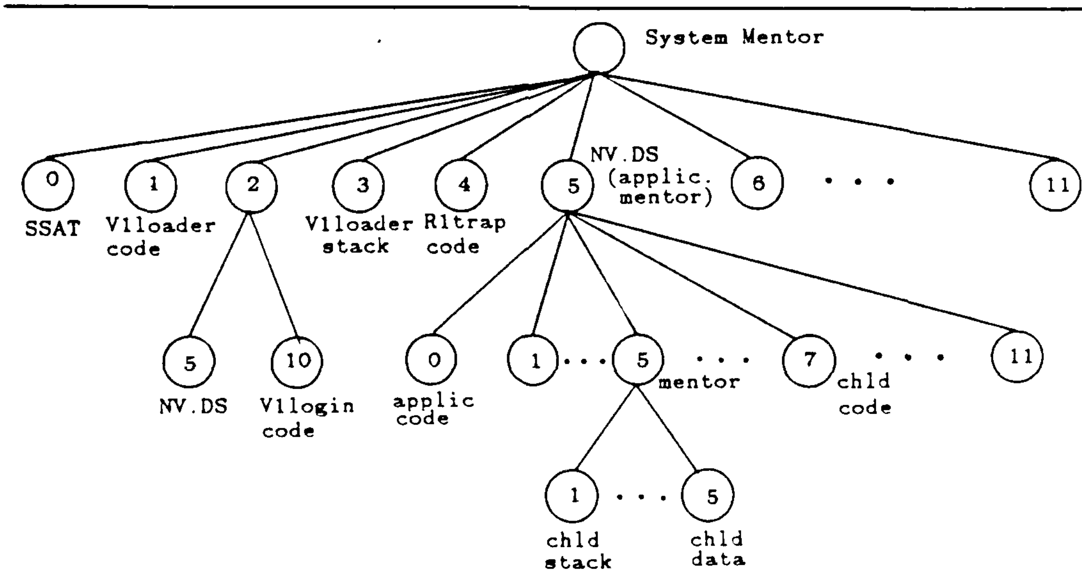


Figure 3.2 Hierarchical Structure including an application

In Figure 3.2 the path 5.7 corresponds to a segment that holds the code of a child application program and must be declared in the Submit file. Entry 5 is the mentor of Entry 7. This creation is static and the path indicated must be known in the application program. On the other hand, the path 5.5.7 is used to hold data and will be created dynamically during execution of the application program. A more complete explanation is presented later in this chapter and in Chapter IV.

C. I/O DEVICE ASSIGNMENT

The process of "Attaching Devices" must be accomplished before an I/O device becomes "known" to the system. It involves assigning a logical process to the I/O device so that the device then is known by its process. The process then contains the device drivers. This step is necessary before any I/O operation. A device can be declared either as a Read *(input)* or *Write (output)* device, as part of the information provided to the **Attach** primitive call into GEMSOS [Ref. 9:pp. 39-41]. A device can be attached to only one process at a time, an error will occur if an attempt is made to attach a device more than one time.

The inverse step is called **Detach** devices, in which the associated process is eliminated and the device becomes available for further assignments [Ref. 9:p. 42].

D. PROCESS CREATION

This section describes the steps that should be considered when creating a process. One process is created from another. The "creator" process is known as the **parent** and the created process is known as the **child**, also having its own unique identifier. The child process receives its fixed amount of resources from the parent. subtracting from the parent's overall resources. A process is a collection of segments known to the process. The segments are managed using a set of primitive functions or "calls" provided by the system. An address space is created to hold a segment. The application programmer must make use of GEMSOS primitives in creating and using this address space.

The sequence of steps that must be followed in order to create a process. starts with the creation of a segment in the address space using the resources available on secondary storage. The primitive **create** is called. resulting in the creation of the address space for this segment. The next step links the space created with a specific process that will use it. In other words. a recognition of this segment is performed in which the process makes the segment known to itself by entering it in the next available entry in its local description table (LDT). The result is the identification of this address space by a number that is called **segment number** which will be used when the segment is referenced. The primitive used is **makeknown**. The result is the identification of this segment for the process and to the system.

The last step is related to the utilization of this segment: a segment must be in main memory in order to be used. This function is performed using the primitive **swap-in**. which produces the loading of the segment from secondary storage into main memory.

When the segment is no longer needed by the process. i.e.. process completed execution, an inverse action must be performed in order to release the space used by the segment. As in the steps declared above, a logical sequencing must be followed, starting with the release of the memory used by the segment. This is performed by the **swap-out** primitive in which the segment is written back out to secondary storage. The next step is the elimination of the association segment-process. Elimination of a segment from a process' address space is performed by the **terminate** primitive: the association is broken, and the entry number used in the LDT of that process is available again.

The total removal of a segment from the system address space is accomplished by using the **delete** primitive; the result is the removal of the segment from the system and process local name space, and the returning of its address space back to the system resources. The steps necessary to create a process are indicated in the Figure 3.3.

1. Create / Makeknown Segments Module

A process needs a minimum of two segments, a code segment and a stack segment, in order to be created. The code segment for a process is declared in the Sysgening Submit file and is created automatically by the system during the system generation (statically).

The stack segment, as well as any additional segments, are created in the user's program (dynamically) by making the appropiate operating system calls for **Create** and **Makeknown**. These calls will be explained later in this chapter.

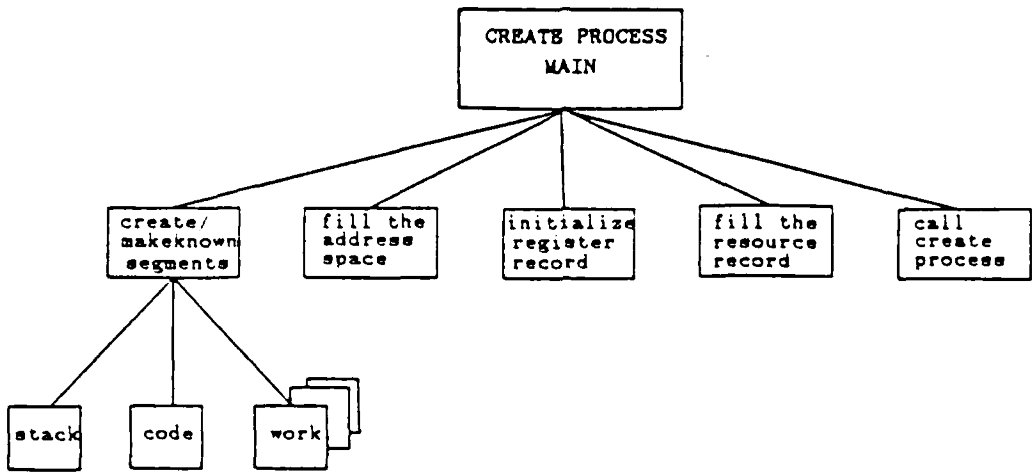


Figure 3.3 Process Creation Main Modules

2. Address Space Specification Module

The address space specification contains a list of the segments that will be passed by the parent process to the new process (child). In addition, the attributes of each segment to be passed are loaded into this module. The address space specification of a process is composed of 5 segments : a stack segment, a code segment (both are compulsory), 2 free segments (can be used as application mentor and for holding data) and a trap segment that is automatically created (it is declared in the submit file) and handled by the system. It holds information for GEMSOS that will be used when a user trap fault is detected. These segments correspond to entries 20 thru 24 of the Local Description Table to be associated with each process.

3. Register Record Module

This module performs the initialization of the register record which defines the state in which the program will begin its execution. The register record contains the following fields [Ref. 9:p. 27] :

1) Instruction pointer (ip) .- Specifies the code offset address.
2) Stack pointers :
   - SP .- The initial stack pointer for the new process.
   - SP1 .- Points to the base of the ring 1 stack segment.
   - SP2 .- Points to the base of the ring 2 stack segment (if one is used).

4. Resource Record Module

In this module the new process (child) attributes are declared. and the amount of resources that the parent process will have to provide to the new process are defined [Ref. 9:pp. 27-28]. The resources received by the child process are subtracted from those of the parent. The amount of resources needed by a child will depend of the kind of application that a child will perform. The resources involved are [Ref. 9:pp. 27-28] :

- Amount of memory (blocks) that the child is allowed to swapin.
- Maximum number of processes that the child is allowed to create.
- Total number of segments that the child will have.

In addition to the resources. a child process number must be declared that uniquely indentifies this child. Also the child's access class must be specified which must be within the parent's range. The resources passed to the child process are recovered by the parent when the child finishes its execution and **self deletes**.

5. Create Process Module

The primitive **Create_Process** is called, resulting in a new process being created. A success code is returned by the operating system to indicate success or failure for this operation. The parameters passed by this module are the record description of the process to be created (r1_cp_struct) and a variable called "success" that will hold the execution's result of this primitive. The application programmer must fill all the fields related to the characteristics, attributes and resources that the new process will have. A description of the record "r1_cp_struct" is given in the library AGATE.LIB provided by Gemini Computers Inc., and in [Ref. 9:pp. 24-28]. Error codes are explained in [Ref. 9:pp. 84-93]. All the modules described above can be executed in any order before the execution of this module.

E. LOCAL DESCRIPTION TABLE (LDT)

Since the actual use of the GEMSOS primitives requires interfacing to the upper level of the application system under development, special interface routines had to be implemented. The next three sections describe these interface routines. A process has a fixed collection of segments which comprisess its address space: these are known by the process as entries in its Local Description Table (LDT). Each process can have up to 52 segments (from 0 thru 51) known to it at one time. Segments 0 to 19 are used by the Kernel, segments 20-24 correspond to segments pre-defined in the address space definition of the new process [Ref. 9:pp.

26-27]. and segments 25 through 51 are available to the application programmer. This segment distribution is fixed in the LDT and can not be modified. A fatal error will result if a system segment is used for other purposes (segments 0 thru 24).

F. CREATE/DELETE SEGMENT

Because a process is a collection of segments. segment creation is an important step that should be considered during process creation. Each segment has its own attributes which are specified in a **Create_seg_struc** record: this record must be declared before calling the primitive **Create_segment**. The segment is created with the specified attibutes. and the addition of a new branch in the hierarchical structure is made [Ref. 9:pp. 14-15]. The inverse action is the **Delete_segment** call. where a segment created previously is removed from the hierarchical structure. this call should be performed when the specified segment is no longer needed [Ref. 9:pp. 16-17]. This call must be used when a process finishes its execution because the applicable segments are not automatically removed.

G. MAKEKNOWN/TERMINATE SEGMENT

The **Makeknown_segment** call adds the specified segment to the address space of the calling process (including the segment number in its LDT table) [Ref. 9:pp. 17-19]. Like all the primitives it has its own record **Mk_kn_structure**. which must be initialized with the pathname that the segment will use in the hierarchical structure. Complete details are provided in Chapter IV. The inverse step eliminates the pathname created in the makeknown process and also frees the segment number from the LDT table. This primitive is **Terminate_segment** and it is described in [Ref. 9:p. 20].

H. SWAPIN/SWAPOUT SEGMENT

A segment is created in secondary storage by first utilizing the primitive **Swapin_segment**, to provide main memory space for the new segment, and the writing the new segment out to secondary storage by utilizing **Swapout_segment** primitive. This function call writes any segment currently stored in main memory out to secondary storage [Ref. 9:pp. 21-22], releasing the main memory.

I. SYNCHRONIZATION

Synchronization among processes is maintained by the use of eventcounts and sequencers [Ref. 10,pp. 115-124], which are maintained in segments used to synchronize processes. The segments used must be common to all the processes involved in the same synchronization. An eventcount is maintained by an integer counter under control of the cooperating process. Completion of an operation (event) is signaled by incrementing the eventcount. The updated eventcount provides a signal to a waiting process that the operation which it has been waiting for is complete. The primitives used are described in [Ref. 9:pp. 33-37].

IV. SYSTEM DESIGN

A. INITIAL DESIGN

1. Objective

The initial objective of the design was to build the upper levels of an operating system based on the Kernel provided by GEMSOS, which would provide multiuser mass secondary storage capability, in a multilevel, secure internal environment. User access through terminals was initially to be without physical security barriers, and terminal access levels determined dynamically based on user access levels. I/O device servicing was to be interrupt driven. Figure 4.1 shows the initial design containing 3 main modules that compose the upper levels operating system : Basic Input/Output (BIOS), Console Command Processor (CCP), and Basic Disk Operating System (BDOS). Implementation of these modules duplicates the same organization used in CP/M or DOS Operating Systems. Secondary storage is to be by segments, providing a "one level" storage system, i.e., no file concept. Security is provided by the GEMSOS operating system.

2. Initial Design Constraints

One major limitation to the above design was the restricted way in which GEMSOS handles interrupts. I/O interrupt handlers currently can not be used from ring 2 or 3 levels, and as such can not be developed in the BIOS module.

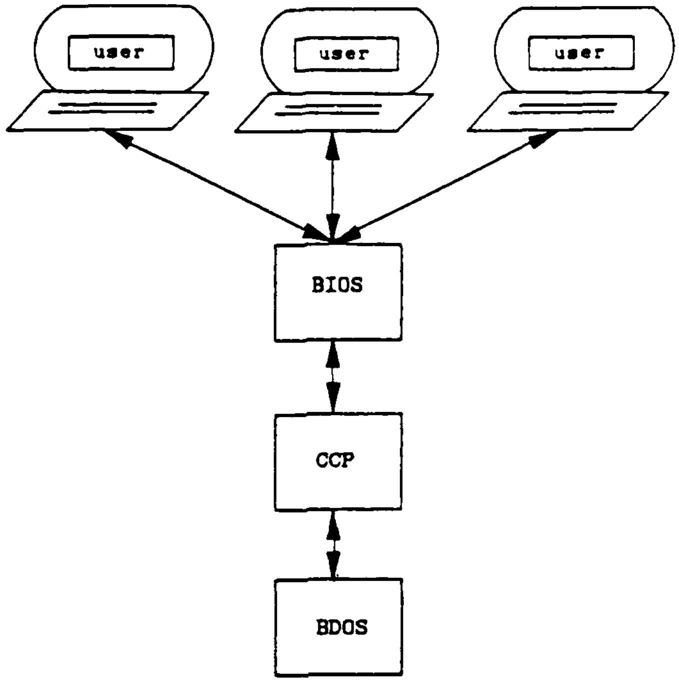


Figure 4.1 Initial Design

Future versions of GEMSOS from Gemini Computers Inc. will resolve this constraint. but the efforts in this thesis were restricted to programmed I/O type of device handling.

The second limitation was related to the number of users that the BIOS module can handle. Since the communications are performed through a RS-232 interface board with 8 ports. and each user needs one physical port (read/write). the maximum number of users is 8 (without considering a device for the main application program). This limitation still exists in the Naval Postgraduate School system configuration.

B. COMPROMISE DESIGN

1. Objective

The primary objective is the same as declared in the initial design, except that the BIOS module is replaced by a dedicated program to handle each terminal, instead of several terminals handled by one program. Each of the programs are identical routines which operate at a **System high** access level to identify a new user through the terminal, and create a corresponding access level process to which the terminal is then "attached". Figure 4.2 shows the actual design used.

With this design, Gemini's capabilities of multiprocessing can also be taken advantage of. Each user can have a virtual processor attached to itself and work in a multiprogramming configuration, limited only by the resources available (processes, segments, memory, ports, etc.). Considering the NPS system, it is possible to emulate multiprocessing with a single processor in which GEMSOS multiplexes the processes using synchronization methods. The distribution of the system resources among individual users can be dynamically assigned/reassigned based on the needs of the user. The amount of resources assigned to the users must not be greater than the resources that the complete system has.

2. Actual Design Constraints

As previously mentioned, the number of users is limited to 8 because of the NPS system configuration.

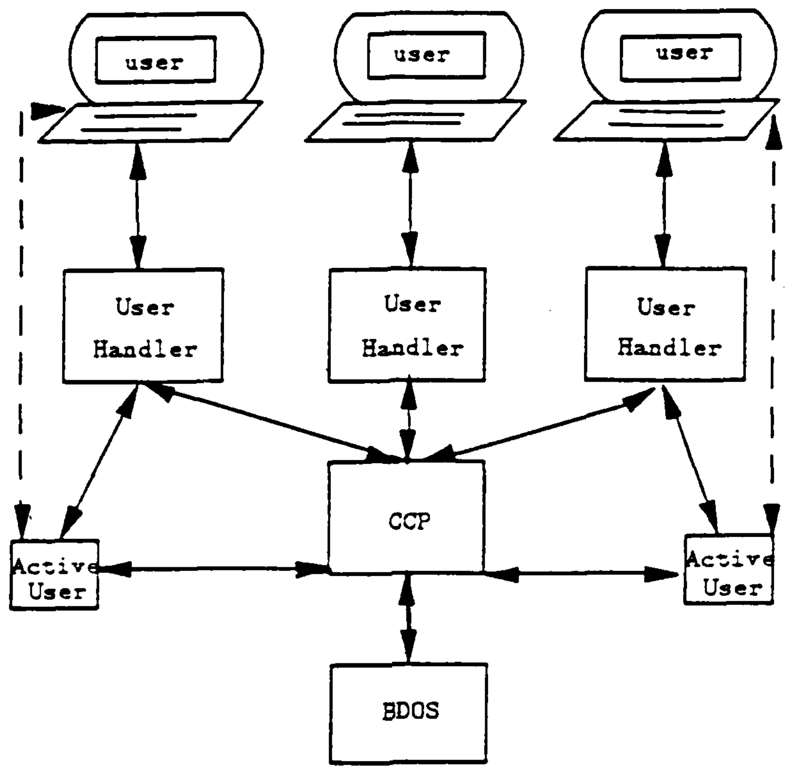


Figure 4.2 Actual Design

The second limitation is related to the size of the LDT. i.e., the number of entries that the application programmer is able to use. If a process nominally needs 3 segments (stack, code, and data) this means that not more than 9 processes can be created in the main application program. Considering the actual design in which a process creates another process, it means that for each user process six segments are needed. Since there are only 27 segments available in the LDT for each process, then the maximun number of processes is 4 (3 users and the BDOS process). By including the data segment in the stack segment, only 4 segments are needed for each process. the result is a maximun of 5 users and the BDOS process.

An additional limitation comes from the fact that there are no developed modules to handle the hierarchical structure or to handle the LDT table of each process. This means that the next available entry or segment number in the LDT or the next entry related to a segment mentor are not dynamically provided by the system and must be obtained in some way. Instead of designing and implementing these modules. temporary modules were designed to create parameters that were useful to the system. A complete set of modules was provided by Gemini Computers Inc. when the system was delivered, but these were coded in Pascal MT+ and the implementation language for this research was Janus/Ada. Conversion from Pascal MT+ to Janus/Ada was one alternative but there was no documentation of the function and structure of these modules resulting in the creation of parameters instead of conversion.

These temporary modules create parameters statically for those that the system should provide automatically. Using these temporary modules it is possible to evaluate system functions and observe how the the hierarchical structure is maintained.

3. Hierarchical Structure Design

Figure 4.3 shows the complete structure design of the system. considering the segments needed to work with 3 users. The numbers indicated inside the circle represent an entry number in the mentor's LDT (maximum 12 segments).

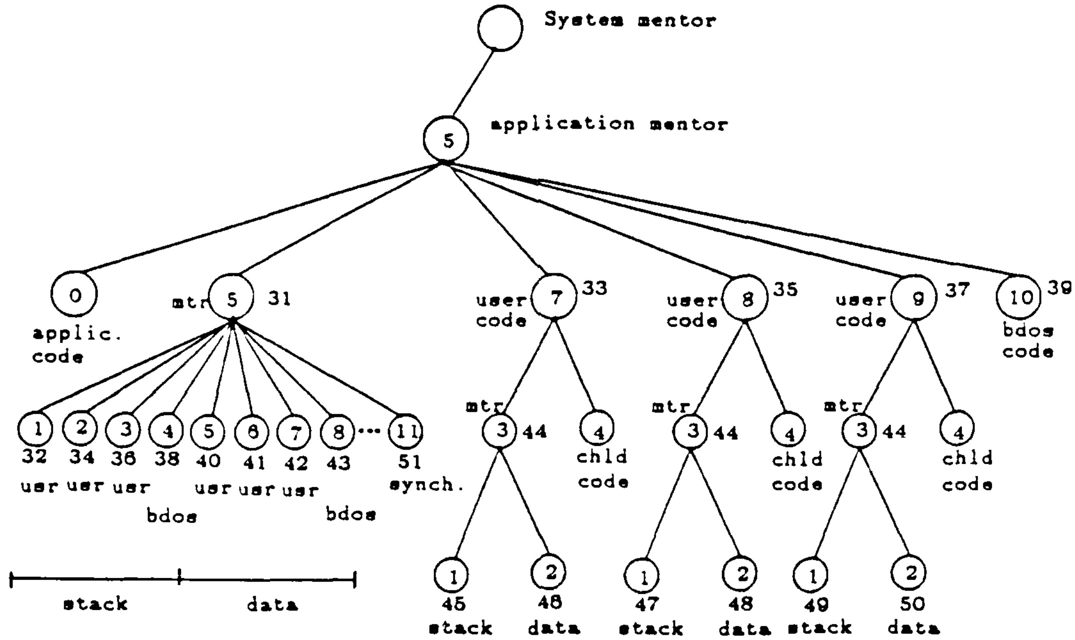


Figure 4.3 System : Hierarchical Structure

and the numbers shown outside the circle represent the process-local segment numbers of the CCP application program. They are used to identify each segment. This can be seen in Figure 4.4. The hierarchy shows the segments 33.35.37.39 as segments that hold the code developed to support the application program in each specific level : User Handler processes and the BDOS process. These segments are specified in the Submit file and are created in the sysgening process together with those segments needed to hold the code of the three Active Users processes. The segments 32.34.36.38 are used to hold the stack segments. and the segments 40.41.42.43 are used to pass information between processes. The segment 51 is used by CCP to effect synchronization with the other processes.

seg # / ntry	user hdlr 1	user hndlr 2	user hdlr 3	bdos process	user applc.1	user applc.2	user applc.3	segment name
0	32	34	36	38	45	47	49	STACK
1	33	35	37	39	--	--	--	CODE
2	51	51	51	51	51	51	51	SYNCHRONIZAT.
3	40	41	42	43	46	48	50	DATA
4	--	--	--	--	--	--	--	TRAP

Figure 4.4 Address Space Specification for each user process

awaiting the eventcount of this segment until some other process advances it, thereby letting the CCP execute its functions.

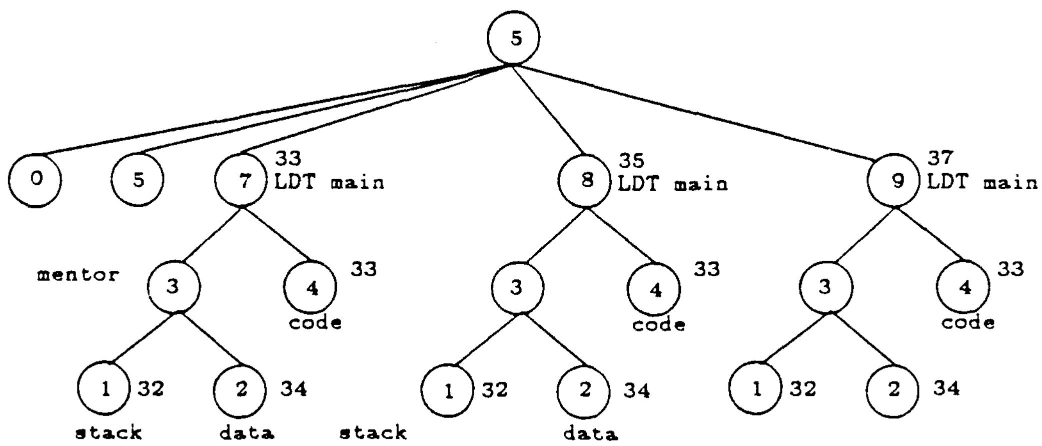


Figure 4.5 Hierarchical Structure : User Processes

A unique pathname identifies each segment in the hierarchical structure. An example of this concept is shown in Figure 4.5. This tree represents the user handler application code segments and their segment numbers in each process.

Although the segment numbers are the same (32.33.34). they differ by the pathname associated with them. i.e.. pathname 5.7.3.1 (user1) pathname 5.8.3.1 (user2). and pathname 5.9.3.1 (user3).

C. SYSTEM SOFTWARE DESIGN

1. Application Programs Design

The design was divided into 2 types of programs. applicative programs (Main Application program CCP. User Handler program. Active User application program. and BDOS application program) and utility programs (common procedures and functions). Structured programming techniques were used to develop the application programs. These techniques are well supported by the implementation language selected. Janus/Ada. Figure 4.6 shows the modules supporting the Main Application program (CCP) that are used to manage the whole system.

These modules are :

1) Load parameters
2) Create segments
3) Create processes
4) Synchronize processes
5) Delete Processes
6) Terminate segments
7) Delete segments

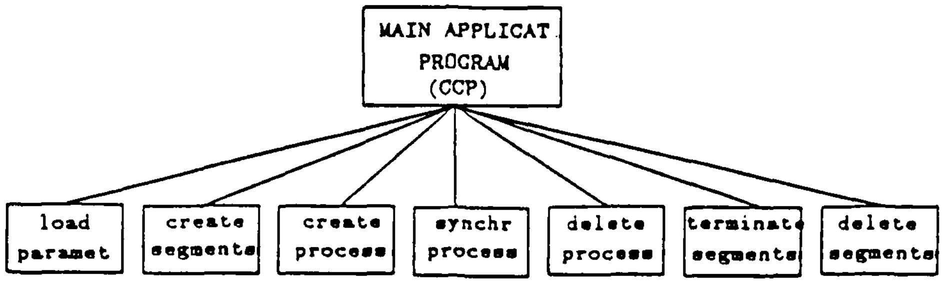


Figure 4.6 Main Application Program

Each module was developed as an independent procedure or function, requiring, in some cases, additional subdivisions because of the high complexity of the module. These modules represent upper level interfaces to GEMSOS system calls.

a. Load Parameters Module

Figure 4.7 shows the table of parameters created by this module. In addition it creates another table with segment numbers that will be used to synchronize the main application program (CCP) with the Active User programs.

b. Create Segments Module

This module creates the segments necessary either to represent some part of the hierarchical structure or to perform synchronization among processes. The size of these segments is fixed (1 byte) because they only function as mentors of other segments or as synchronization segments. A synchronization segment is created as a common segment, which must be known for all the processes in the

		user handler 1	user handler 2	user handler 3	BDOS
STACK	mentor	31	31	31	31
	entry	1	2	3	4
	number assigned	32	34	36	38
CODE	mentor	seg(2) 22	seg(2) 22	seg(2) 22	seg(2) 22
	entry	7	8	9	10
	number assigned	33	35	37	39
DATA	mentor	31	31	31	31
	entry	5	6	7	8
	number assigned	40	41	42	43

Figure 4.7 System Parameters

application program. This is necessary because the execution of the main module CCP must be accessible to all processes communicating with the CCP to perform some operation.

The segments created are :

1) Function : mentor of stack and data segments of the User Handler processes
   Mentor : initial segment (2) system segment
   Entry : 5
   Classif : unclassified
   Number : 31

2) Function : synchronization
   Mentor : segment 31 (created previously)

Entry : 11
Classif : top-secret
Number : 51

c. Create Process Module

This module contains a loop that is executed 4 times which creates three User Handler processes (3) and the BDOS process. All of these processes will have multilevel access classes to handle users with different levels of clearance.

This module is sub-divided into sub-modules that are easier to test and understand. The sub-modules are :

1) Create and Makeknown segments
2) Fill the Address Specification
3) Initialize the Register record
4) Fill the Resource record

An explanation of each module was provided in Chapter III. The main point here is that the process will be created using the paramenters loaded previously and each process will receive the following resources :

Memory : 100 bytes
Segments : 300 segments available
Process : 1 (max number of processes that can create)

d. Synchronization Module

The synchronization used can be cataloged in one of four types :

1) Main Application Program-User Handler Process. This is performed in two ways :

- After process creation the User Handler Process communicates that it was created and returns control to CCP.

- During process synchronization the User Handler Process will communicate to CCP that a User (terminal) is active or inactive.

2) Main Application Program-Active User Process. This is performed when the Active User created by a User Handler process sends a message (command + information) to CCP in order to execute some specified operation. i.e.. The CCP performs this command (calling BDOS if necessary) and returns a result to the Active User.

3) Main Application Program-BDOS. This synchronization is performed when the Active User executes a command that requires BDOS participation. such as a read/write segment to secondary storage. CCP receives the message from the user and directs it to BDOS. When BDOS executes the command. it returns the result to CCP which in turn directs the result to the Active User.

4) User Handler Process-Active user Process. This is performed when the Active User is created and communicates that it was created. returning the control to the User Handler. The same communication is performed when the Active User finishes execution (entering "bye").

When the communication is between CCP and a User Handler or an Active User. the CCP must recognize which user sent the message in order to return the result. The synchronization is obtained using the following segments :

1) Stack Segment. To synchronize the execution of that process.

2) Data Segment. To determine which user was activated. The CCP stores the previous value of the data segment of each process. When User Handlers or Active Users send a message. it advances the eventcount (also the synchronization segment eventcount) then CCP compares this value against the value stored previously. If the result is different it means user activated. otherwise CCP checks the eventcount of the next user until an active user is found.

3) Synchronization Segment. To synchronize the execution of the CCP. All the processes know this segment in order to activate the CCP execution. advancing the eventcount of this segment. When CCP finishes execution. it advances the eventcount of the process that activated it previously and then waits until another process calls it again.

The CCP knows the synchronization segments used by each process since they are specified in the system parameters. The segments used are :

	STACK	DATA
user handler 1	32	40
active user 1	45	46
user handler 2	34	41
active user 2	47	48
user handler 3	36	42
active user 3	49	50

The synchronization mechanisms used here are **Await**. **Advance** and **Read_eventcount**. as provided in GEMSOS.

e. Delete Processes Module

This module will delete the processes created before. It is executed when users enter "bye". In addition to this, it will also terminate and delete the segments created in the process creation (stack and data). and will terminate the code segment. The success of this module depends on the previous self deletion of the User Handler process.

f. Terminate Segments Module

This module terminates the segments created previously (mentor and synchronization segments). The result is that segments numbered 31 and 51 are now available in the LDT.

g. Delete Segments Module

This module returns the space used by the segments terminated before, and marks free the entry numbers used to create these segments.

2. User Handler Application Program

The design is like that of the CCP design. The differences are related to the way the modules perform internally. The function of this application program is to create a single access level Active User according to the user clearance level determined during the Logon process. It is also divided into the following modules :

a. Load Parameter Module

This module creates a table with parameters that are needed in the Active User process creation. The parameters are the same for all users' processes since each User Handler has its own LDT table. This means that they have independent segments. Again, Figure 4.5 shows the structure and numeration of each segment.

b. Create Segments Module

This module creates the segment that will be used as mentor of the Stack and Data segments for the Active User process. One difference between this module and the Create_segment module in the Main Application program (CCP) is that it does not create a specific synchronization segment. Another difference is related to the mentor used to create the User Active process. The User Handler program's code was used, because it is unclassified and can satisfy requirements of security.

An unclassified segment has minimum access class constraints and can be mentor of classified segments. As such the segment that holds the User Handler code was used since it is unclassified and satisfies security requirements. The inverse case occurs when a classified segment is used. It can only be mentor of those segments with equal or greater classification. This means that segments with less access class cannot be created, limiting the participation of users with no classification level. In a multilevel system, the mentor segments must be capable of handling different kinds of users and segments associated with those users.

c. Create Process Module

This module creates a single access level Active User Process (only one). The access level is determined when the user enters the system. The difference between this module and the one contained in the Main Application program is the resources assigned to the created process :

Memory	:	60 bytes
Segments	:	100 segments available
Processes	:	0 (cannot create child processes)

d. Synchronization Module

The synchronization used was explained previously; the segments used are :

1) Code Segment. To synchronize the execution of User Handler Process.
2) Stack Segment. To synchronize the execution of the Active User.

This synchronization takes place when the User Handler process releases the attached terminal. creates and activates the Active User process (advancing its stack eventcount), and waits until the Active User (child) finishes execution (entering "bye" and self deleting). When this happens, it terminates and deletes the segments created to support the execution of the Active User (stack, data, mentor segments), and communicates to CCP that this user is inactive.

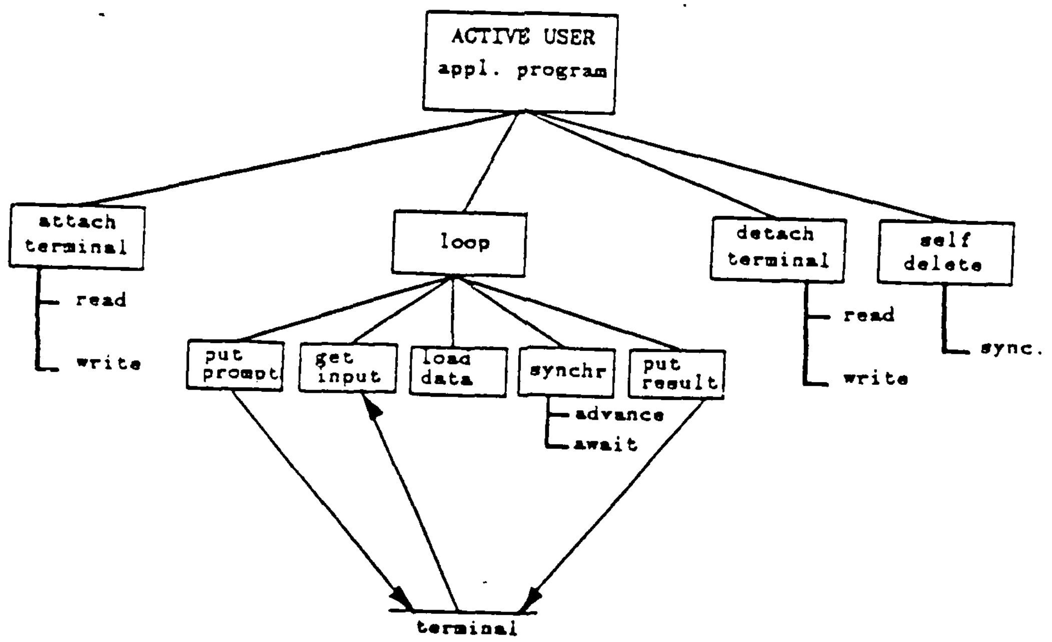


Figure 4.8 User Active Application Program Modules

3. Active User Application Program

This program was designed following structured programming techniques. Figure 4.8 shows the modules that compose the Active User application program. These modules are :

1) Attach terminals (read - write)
2) Loop
   - Prompt the user
   - Get user input
   - Load data segment
   - Synchronize
   - Put result
3) Detach terminals
4) Self delete

Following the same structure used in the description of the Main Application program and the User Handler Application program, the modules indicated above are described individually :

a. Attach-terminal Module

This module assigns a terminal to the Active User process, the object being to allow the user to perform I/O operations. There is a specific **Attach_device** call for assignment of read and write.

b. Loop Module

This module handles the main process. It starts with the input entered by the user, and finishes when the user receives a result for the input entered. The intermediate steps necessary to get the output result from CCP are :

1) Load Data Segment. The segment created to pass data.

2) Synchronize with CCP. Data segment is passed to CCP and the synchronization segments are activated (advance synchronization and data segments eventcounts; await stack segment eventcount).

c. Detach_terminal Module

This module detaches the device previously attached, returning control of this device to the Operating System, making it available for future assignments in other processes. There must be a balance between Attach and Detach calls, otherwise a system error will occur.

d. Self_delete Module

This module holds the ending of a child process. Self_delete is required if the next step is delete the child (Active User) process from the address space of the parent (User Handler). When the delete process is complete, the parent recovers all the resources that were assigned to the child in the Create_process step. In addition to the Self_deletion call, a segment number must be specified in order to perform the synchronization. This is due to the fact that when the process finishes, it automatically Advances the segment eventcount indicated in the Self_delete call.

V. IMPLEMENTATION

A. GENERAL DISCUSSION

Implementation of the model was designed considering one application program for each process. The following programs were developed :

1) Main Application Program- CCP	(Appendix A)
2) User Handler Application Program	(Appendix B)
3) Active User Application Program	(Appendix C)
4) BDOS Application Program	(Appendix D)

To this end, it was necessary to construct the following support programs :

1) Primitive Calls
2) Auxiliary Functions

1. Primitive Calls

The object of this program is to be used as an interface between the GEMSOS primitives at the Kernel level of the system and the application programs. All primitives use a record structure initialized by the programmer before calling for the specific operation. The set of programs developed to perform these functions are called "PROCE": they were built by modifying the demonstration program provided by Gemini Computers Inc. and adapting them to the requirements of the proposed model.

The program "PROCE" has two extensions: "PROCE.LIB" contains the specifications that can be visible to the user, and "PROCE.PKG" contains the code developed to handle these specifications. T is is an ADA feature that helps to reinforce security aspects (Information Hiding Principle).

The procedures developed in these modules are :

PROCEDURE	PRIMITIVE SUPPORTED
CR-SEGMENT	CREATE-SEGEMENT
DL-SEGMENT	DELETE-SEGMENT
MK-SEGMENT	MAKEKNOWN-SEGMENT
MAKEKNOWN-SYNCH	Used to makeknown the synchronization segments (CCP-to-Active User)
TERMINATE-SYNCHR-SEG	Terminates the segments indicated above
FILL-INIT	Fills the resources record needed by Create-process primitive
CR-PROCESS	CREATE-PROCESS

The program listing is contained in Appendix E. Those GEMSOS primitives not requiring record initialization (i.e.. **Terminate-segment** and all the primitives used for synchronization) were not included. The primitives **Attach-device** and **Detach-device**. were separated into the Auxiliary Functions program because they may be called in future applications that may not need the use of the other primitives declared above.

2. Auxiliary Functions

This program contains additional data structure needed by the main application program. i.e.. command record. password's record description. constants used. etc.. It also contains procedures and functions to perform I/O operations (read and write), and functions to create parameters replacing the modules that the system needs but were not delivered in the NPS Gemini package

(The LDT entry allocation procedure for example). These were described in the design constraints of Chapter IV. As in the previous program. some procedures and functions provided by Gemini Computers Inc. were used as a basis to construct this program segment, with the addition of code and records required to support the current research. This program. called "FILES", has two extensions. "FILES.LIB" contains the specifications and records. and "FILES.PKG" contains the code developed.

The procedures and functions included are :

NAME	TYPE	DESCRIPTION
GET-STR	Procedure	Gets a string from the terminal
PUT-STR	Procedure	Displays a string in the terminal
PUT-DEC	Procedure	Displays a number in the terminal
PUT-SUCC	Procedure	Displays a string and a number
GET-INPUT	Function	Gets an input string from the terminal and echoes the input if the echo option is on. It also converts all the input to lower case
ATTACH-TEW/TER	Procedure	Supports the primitive ATTACH DEVICE specifying if the device is to read or write
LOAD-PARAM	Procedure	Produces a table of parameters with information needed by the main application program (segment number. entry. mentor)
LOAD-ACCESS-CLASS	Procedure	Produces a table with the security access level depending on the user level
LOAD-PARAM1	Procedure	Produces a table of parameters with information needed by the user handler program (segment number.entry.mentor)
LOAD-CHILD-ACTIVE	Procedure	Initializes the Active User record to false. It lets the CCP load the Active User segments each time a false record is found

LOOK-FOR-LEVEL	Procedure	Simulates the Logon process, loading the access class of the user depending on Username and Password
CONVERT	Procedure	Assembles the command line using the input message typed by the user

The program listings are contained in Appendix F.

B. IMPLEMENTATION CONSTRAINTS

Two factors dictated splitting the implementation into several steps thus making the system easier to develop and test. These factors were :

1) Lack of System Documentation and prior experience.

2) Time required to develop, implement, integrate and test.

At the time this research started, sufficient information about the system did not exist. As such numerous system "quirks" posed additional time delays in the implementation process. Upper level interfaces to GEMSOS had to be implemented, at times by experimentation. Program development using an unfamiliar set of procedures and new functions is error prone, requiring programming tools that are still not available in this machine. Another implementation constraint was the time required for program development.

A main factor related to the speed of development is the fact that a program, in addition to compiling and linking, requires a special process, called "Sysgen" prior to execution. Sysgening takes longer than 10 minutes each time, even if only a single line was changed. Since debugging tools were limited, it often took several attempts to find a mistake or the exact way of performing a specific operation. As previously described, the "sysgen" process has the functions of creating and formating logical volumes in secondary storage, and creating a bootable **Gemini System Segment Structure** on formatted volumes. This second function is called each time a program must be loaded to execution [Ref. 8:p. 1]. The use of the system program (sysgen) is explained in [Ref. 8:pp. 8-18].

C. IMPLEMENTATION STEPS

The basic approach starts with a model using the demonstration program provided by Gemini Computers Inc. The primary steps followed were :

1. Single User, Single Security Access Level

This step established the ability to create a child process and to establish communication between parent and child processes. The main issue here was the stack size. Its size had to be determined experimentally (AFFF hex), since it is not clear as to the correct way to measure the size. The size of the stack is determined by the following constants :

stack-size	= vector-size	+ segment-manager-size	+ constant	
	=	4 bytes	76 bytes	AFFF bytes

The size of the last constant was derived empirically, and is apparently a combination or the amount of memory needed to create a child process and the number of processes that can be activated simultaneously in the system.

2. Several Users, Single Security Access Level

This step proved the creation and synchronization among several processes. The key points were :

a) The hierarchical structure of the system required special procedures for handling. A procedure was created to dynamically determine the next segment available in the system. This procedure was written to associate the next segment number to be used in process creation, and the entries used by each segment. The resulting table of parameters created by this procedure was previously shown in Figure 4.7.

b) A synchronization method among processes was needed, which included the structure needed to determine which process was activated (two synchronization segments for each process).

c) Communication between processes was developed (passing information). The procedure **Move_bytes** is considered to pass information from one process to another, an example and further explanation is provided in [Ref. 9:pp. 5-6].

3. Several Users, Multilevel Security Access

This step introduced the security constraints needed to work in a multilevel secure system, the key points were :

a) Creation of a Child process (Active User) by another child process (User Handler) previously created by a main application program (CCP). This addresses the resources passed by the parent process. A balance was achieved between the resources passed by CCP when it creates a **User Handler** process, and the resources passed by the **User Handler** process to an **Active User** process during its creation. In other words the following constraints were applied :

(User Handler (1 + 2 + 3) + BDOS resources) < CCP resources

Active User resources < User Handler resources

b) Development of three kinds of synchronization : between Child (Active User) and Parent (User Handler); Grandchild (Active User) and Grandparent (CCP); Parent (User Handler) and Grandparent (CCP). Different segments

		pathname
	0 - 19 kernel	
	20 - 24 address specif.	
	25 - 30 not used here	
31	users' mentor	5,5
32	stack user 1	5,5,1
33	code user 1	5,7
34	stack user 2	5,5,2
35	code user 2	5,8
36	stack user 3	5,5,3
37	code user 3	5,9
38	stack BDOS	5,5,4
39	code BDOS	5,10
40	data user 1	5,5,5
41	data user 2	5,5,6
42	data user 3	5,5,7
43	data BDOS	5,5,8
44	mentor chld users	5,7,3/5,8,3/5,9,3
45	stack chld user 1	5,7,3,1
46	data chld user 1	5,7,3,2
47	stack chld user 2	5,8,3,1
48	data chld user 2	5,8,3,2
49	stack chld user 3	5,9,3,1
50	data chld user 3	5,9,3,2
51	SYNCHRONIZATION	5,5,11

Figure 5.1 Main Application Program LDT

were used to synchronize the execution of Active User and User Handler from those used to synchronize Active User with Main Application (CCP) or User Handler with CCP. The synchronization of the Main Application program with all users was implemented through the same segment.

c) Restrictions imposed on segment handling due to security access levels: segments with equal or greater access class must be passed between processes. The security access level is composed of two parts : Compromise (Observe) and Integrity (Modify). This research worked within Compromise constraints. Integrity involves levels in the system's rings, compartmentalization (audit. operator. programmer. etc.). and the concept of ring brackets. In order to keep the model simple. integrity was not considered. The execution of each primitive checks security constraints and if satisfied. the execution continues. otherwise the system aborts for security reasons.

d) Main Application Program Segment Distribution. Figure 5.1 shows the LDT table including all segments needed by the seven processes (3 User handler processes. 3 Active User processes. and BDOS process). It also presents the segments used as mentors of other segments (31 and 44) and the segment used to synchronize the CCP (51).

D. SYSGEN SUBMIT FILE

Appendix F shows the Sysgen Submit File used to define the structure of the application programs developed in this research.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

1. System Operation

The security check starts with the **logon process**. If the operator has a valid username and password that is recognized by the system, then the security level of this operator is adopted for the terminal, otherwise the system continues prompting for the username three more times. Failing a correct response, the system restarts the logon process. The main testing performed was the validation of the segments defined in the submit file. If they were "classified" the operator had to satisfy the security constraints in order to use this application. Otherwise, the system aborted the execution and prompted for a new logon operation.

The application programs work according to the design, dynamically loading the user's security level in the "terminal logon" process. Access is limited to those users recognized by the system and restricted to the use of information based on the user's access level. The interaction between Active User-CCP-BDOS is performed within security constraints (only compromise). The messages passed are "processed" by the CCP and results are returned to the user.

2. System Performance

Because of the NPS system configuration, the performance of the application program is degradated for the following reasons :

- Single processor emulating parallel processing
- Secondary storage consisting of only floppy disks
- Programmed I/O environment instead of interrupts

B. RECOMMENDATIONS

1. Hardware Improvements

A system upgrade should be considered which at least addresses adding a hard disk to the present configuration. This would provide an improvement in the access time required to handle segments and decrease the response time in process creation. In addition, there would be a considerable reduction in the system development time, i.e., the time required to compile, link and sysgen, frequent operations in the development phase.

Additionally, to take full advantage of the machine's parallel processing abilities, more processors, two at least, should be added. Memory expansion would relax many restrictions currently imposed on process creation as utilized in this thesis work.

2. Software Improvements

The current system as delivered was incomplete with respect to the modules necessary to develop application programs in the Janus/Ada language.

Software improvements should include better documentation related to the system and debugging and programming tools for the application programmer.

3. Future Research

As a result of the research presented in this thesis. areas of related research should include the following :

a. Directly Related to this Research

(1) **Inclusion of integrity access constraints (modify) into this current implementation.** In this thesis, system integrity was considered at its minimum level in order to execute the application programs without limitations. Since this is a main issue with respect to the information security, a careful implementation should be developed working in this area.

(2) **Development of interrupt driven environment** in the current design. The initial design using the BIOS module is an event driven system. This design approach was not used due to present hardware limitations as explained in Chapter IV. "Initial Design Constraints".

(3) **Addressing (solving) the issue of a restricted LDT size.** This restriction is due to a limited number of application programs or application program complexity. since an upper bound of 27 segments are available for the user from the bounded LDT of 52 that a process can have.

(4) **Implementation of the "terminal logon" method**. This function is simulated in the actual development through a module that loads the access class of a recognized user. A possible solution would be the conversion of those libraries provided by Gemini Computers Inc. from Pascal language into Ada language.

b. Related to Secure Mass Storage System

(1) **Development of software "crossbar" in the Concentrator** for integration of the Gemini computer into the Naval Postgraduate School Laboratory as a **Secure Mass Storage System (S3)**.

(2) **Implementation of BDOS** using a design for a one-level segmented secondary storage system and a large capacity hard disk (mass storage).

APPENDIX A - MAIN APPLICATION PROGRAM (CCP)

This appendix presents the Main Application Program code. including the modules developed to handle the dynamic sharing of system resources. Instead of the present system consisting of several programs. each containing one module. all modules are grouped into one main program with the indicated modules separated by comments. This program is called "PRMAIN" and is listed below.

```
pragma rangecheck( off ); pragma debug( off );
pragma arithcheck( off ); pragma enutab( off );

WITH agate, agatej, ar1, alib, alibj, strlib, util, proce,
      files;
PACKAGE BODY prmain IS
USE agate, agatej, ar1, alib, alibj, strlib, util, proce,
      files;

--   ***********************************************************

--   Constants used by the program :
--
--    STDIO_W    -->  assigns logical device 1 to write
--    STDIO_R    -->  assigns logical device 0 to read
--    IO_PORT    -->  main program uses port 0 of the RS-232
--    PRBDOS     -->  process number for the BDOS


STDIO_W : CONSTANT integer := 1;
STDIO_R : CONSTANT integer := 0;
IO_PORT : CONSTANT integer := 0;  -- port zero for main
NUMBER_OF_USERS : CONSTANT integer := 3;
NUMBER_OF_PROCESS : CONSTANT integer := 4;
PRBDOS : CONSTANT integer := 4;
NUMBER_PROCESSES : CONSTANT integer := 1;  -- # of childs
MEMORY_AVAILABLE : CONSTANT integer := 130;
SEGMENTS_AVAILABLE : CONSTANT integer := 300;


-- Variables used by main program.

w_class : access_class;              --  security aspects
success : integer;                   --  result
mode    : attach_struct;
mode_r  : attach_struct;
def_off : integer;
def_seg : integer;
r1_def_size : integer;
synchr_seg : integer;
ch_out : comand_line;
ch_in : input_message;
ch_resource : child_resource;
init : r1_process_def;
rd_str : string;
class          : access_class;
mentor         : integer;
entryx         : integer;
seg_mode       : seg_access_type;
seg_number     : integer;
CCP_BUSY       : boolean;
```

```
ch_parameter : r1_parameters;
ch_param : r1_param;
ch_level : user_level;
ch_access_level : level_record;
ch_ch_user_synchro : users_active;
ch_ch_evc_val      : integer;
proces   : integer;
no_active_users : integer;
evc_value : integer;
evc_active : integer;
active_user : integer;
index      : integer;

-- MAIN
BEGIN
    init := get_r1_def();           --  **  this sentence is
                                    --      obligatory

--  ****************************************************************

--   attach serial port for writing.
--   This sentence is optional and is used to display messa-
--   ges provided by the main application program on the
--   screen .

    attach_tew( IO_PORT, STDIO_W );

    lib_set_bracket( 1, 1, 1, init.resources.min_class );

--  ****************************************************************
--  LOAD PARAMETERS MODULE
--
--  This module assigns fixed parameters to each process
--  that will be created. The parameters are mentor number
--  entry number, segment number. They are used to create
--  each segment needed by the process to be created
--  (USER HANDLER). There is a group of these parameters for
--  the stack, code, and data segments


    load_param(init, ch_param);

--  load child actives array are additional parameters
--  used by the main application program to synchronize its
--  operation with the ACTIVE USER processes

    load_child_active(ch_ch_user_synchro);

--  load access class that each process will have. In our
--  case all the processes will be multilevel. Minimum is
--  unclassified and maximum is top-secret
```

```
    load_access_class(init,ch_level);

    w_class := init.resources.max_class;

--  ******************************************************************

--  ******************************************************************
--
--  CREATE SEGMENTS MODULE
--
--  This module creates the segments needed for the
--  following :
--     a.-  Mentor of the stack and data segments of each
--          process and BDOS process. The standard elected
--          was use entry 5 of the mentor intial_segment(2)
--          "application mentor" and call this new segment
--          31 in the LDT of the main
--
--     b.-  Synchronization segment. Its mentor is the
--          segment created in the previous step, entry 11
--          of segment 31 was elected and this new segment
--          was called 51 in the same LDT. The access class
--          is TOP-SECRET

    ch_access_level := ch_level(1);
    mentor      := init.initial_seg(2);
    entryx      := 5;
    class       := init.resources.min_class;
    cr_segment(init,mentor,entryx,class,success);
    if success /= 0 then
       put_succ("success value 02 is",success,w_class);
       put_ln(STDIO_W,w_class,"");
    END  IF;

--  makekwown this segment with number 31
    seg_mode  := r_w;
    seg_number := 31;
    mk_segment(init,mentor,entryx,seg_number,
                 seg_mode,success);
    if success /= 0 then
       put_succ("success value 01 is ",success,w_class);
       put_ln(STDIO_W,w_class,"");
    END IF;

--  create synchronization segment   (max access class)

    class := ch_access_level.max;
    mentor := 31;        -- segment 31
    entryx := 11;
```

```
    cr_segment(init,mentor,entryx,class,success);
    if success /= 0 then
       put_succ("success value 23 is ",success,w_class);
       put_ln(STDIO_W,w_class,"");
    END IF;

--  makeknown this segment with number 51

    seg_mode := n_a;
    seg_number := 51;
    mk_segment(init,mentor,entryx,seg_number,
               seg_mode,success);
    if success /= 0 then
       put_succ("success value 24 is ",success,w_class);
       put_ln(STDIO_W,w_class,"");
    END IF;
    synchr_seg := seg_number;

--  swapin this segments

    swapin_segment(synchr_seg,success);

    if success /= 0 then
       put_succ("success value 25 is ",success,w_class);
       put_ln(STDIO_W,w_class,"");
    END IF;

--  ************************************************************************

--  ************************************************************************
--
--  CREATE PROCESS MODULE
--
--  This module creates 4 processes
--     (3 user handler and 1 BDOS)

      put_ln(STDIO_W, w_class, "BEGIN PROCESS CREATION");

--  START CREATING EACH PROCESS IN THE SYSTEM
--
--    process 1     ====>   TERMINAL HANDLER
--    process 2     ====>   TERMINAL HANDLER
--    process 3     ====>   TERMINAL HANDLER
--    process 4     ====>   BDOS HANDLER
--
      index := 1;
      while index <= NUMBER_OF_PROCESS  LOOP
--
        ch_parameter := ch_param(index);

--  all the processes will have multilevel access class
```

```
          ch_access_level := ch_level(1);

--   load the resources that the child will have
--
--        memory -->  100   (converted to B24 format)
--        segments >  300
--        processes   1 (max number of proc. that can create)

        ch_resource.memory := MEMORY_AVAILABLE;
        ch_resource.segments := SEGMENTS_AVAILABLE;
        ch_resource.processes := NUMBER_PROCESSES;

        read_evc(synchr_seg, evc_value, success );

        cr_process(init,ch_parameter,ch_access_level,
                   index,ch_resource,synchr_seg, success);

--   synchronize each time to let the new process displays
--   its message (word USER)

          await(synchr_seg, evc_value+1, success);
          index := index + 1;

         end LOOP;
--
--   ****************************************************************

--   ****************************************************************
--    SYNCHRONIZATION PROCESS AND LOOP UNTIL NO USER IS
--    ACTIVE
--
--    This module executes the synchronization among
--    processes, starting with the synchronization with the
--    user's handler processes and then with the active user
--    processes when these have been activated

--     ACTIVATE USER'S PROCESS
--      This step stores the value of the eventcounts in the
--      users' segments, the object is to know which user
--      sent the message

          index := 1;
          while index <= NUMBER_OF_USERS  LOOP

          read_evc(ch_param(index).seg_number_stack,
                   ch_param(index).evn_count, success );
          read_evc(ch_param(index).seg_number_data,
                   ch_param(index).evn_count_data,success );
          advance(ch_param(index).seg_number_stack,success );
          index := index + 1;
```

```
      end LOOP;

--     LOOP UNTIL NO USER IS ACTIVE
--    This loop is the main process's module in the whole
--    system because will be in execution until all the users
--    finish their jobs (enter the word "bye")

      CCP_BUSY := FALSE;
      no_active_users := 0;
      while no_active_users < NUMBER_OF_USERS  LOOP

         if CCP_BUSY then
            CCP_BUSY := FALSE;
         else
             read_evc(synchr_seg, evc_value, success );
             await(synchr_seg, evc_value+1, success );
         END if;

         active_user := 0;
         index := 1;
--
--     determine the user that sent the message comparing the
--     event count value, different value means that user was

         while index <= NUMBER_OF_USERS   LOOP
            read_evc(ch_param(index).seg_number_data,
                    evc_active, success );
            if (evc_active >
                    ch_param(index).evn_count_data) then
                 active_user := index;
                 ch_param(index).evn_count_data :=
                                   evc_active;
                 index := NUMBER_OF_USERS + 1;
            else
                 index := index + 1;
            END IF;
         END LOOP;

--     checks if the activated process came from an user
--     handler or an active user. If came from the active
--     user it means that the variable "active_user" is in 0,
--     otherwise it means that the communication is between
--     a user handler process and the main program (CCP)

         if active_user /= 0 then
             def_seg    := lib_mk_sel(ldt_table,
                        ch_param(active_user).seg_number_data);
             def_off    := 0;
             r1_def_size := input_message'SIZE/8;
```

```
move_bytes(def_seg,def_off,get_ss(),
           ch_in'ADDRESS, r1_def_size);
ch_parameter := ch_param(active_user);
if ch_ch_user_synchro(active_user).active then
    ch_ch_user_synchro(active_user).active :=
                                        FALSE;
    terminate_synchr_seg(init,ch_parameter,
                    ch_in.input_class,success);
    if success /= 0 then
        put_succ("terminate ",success,w_class);
        put_ln(STDIO_W,w_class,"");
    end if;
    if ch_in.input_one = "bye" then
        no_active_users := no_active_users + 1;
    else
        advance(
         ch_param(active_user).seg_number_stack,
                  success);
    END IF;
else
    if ch_in.input_one /= "bye" then
       make_know_sync(init,ch_parameter,
           ch_in.input_class, active_user,
           ch_ch_user_synchro, success);
       ch_ch_user_synchro(active_user).active :=
                 TRUE;
       advance(
         ch_param(active_user).seg_number_stack,
                  success);
    else
       no_active users := no_active_users + 1;
    END IF;
END IF;
ELSE
   index := 1;
   while index <= NUMBER_OF_USERS  LOOP
     if ch_ch_user_synchro(index).active then
       read_evc(ch_ch_user_synchro(index).seg_data,
                ch_ch_evc_val, success);
       if ch_ch_evc_val >
            ch_ch_user_synchro(index).evc_data then
          active_user := index;
          ch_ch_user_synchro(index).evc_data :=
                     ch_ch_evc_val;
          index := NUMBER_OF_USERS + 1;
       END IF;
     END IF;
     index := index + 1;
   END LOOP;

   if active_user = 0 then
```

```
            put_ln(STDIO_W,w_class,"active user error");
          END IF;

          def_seg    := lib_mk_sel(ldt_table,
                 ch_ch_user_synchro(active_user).seg_data);
          def_off    := 0;
          r1_def_size := input_message'SIZE/8;
          move_bytes(def_seg,def_off,get_ss(),
                    ch_in'ADDRESS, r1_def_size);
--
        convert(ch_in,w_class,ch_out);

--  pass the segment to prbdos process
--
          def_seg := lib_mk_sel(ldt_table,
                     ch_param(PRBDOS).seg_number_data);
          def_off    := 0;
          r1_def_size := comand_line'SIZE/8;
          move_bytes(get_ss(),ch_out'ADDRESS,def_seg,
                    def_off,r1_def_size );
--
--  ACTIVATE PBDOS PROCESS
--

          advance(ch_param(PRBDOS).seg_number_stack,
                 success );
          read_evc(synchr_seg, evc_value, success);
          await(synchr_seg, evc_value+1, success );


--   RECEIVED MESSAGE WITH RESULT HAS TO BE PASSED TO THE
--   USER

           def_seg    := lib_mk_sel(ldt_table,
                      ch_param(PRBDOS).seg_number_data);
           def_off    := 0;
           r1_def_size := comand_line'SIZE/8;
           move_bytes(def_seg,def_off,get_ss(),
                      ch_out'ADDRESS, r1_def_size);

--   assemble user message with the result

         ch_in.input_result := ch_out.result;

--
--  pass the segment to user process
--
            def_seg    := lib_mk_sel(ldt_table,
                ch_ch_user_synchro(active_user).seg_data);
            def_off    := 0;
            r1_def_size := input_message'SIZE/8;
```

```
                move_bytes(get_ss(),ch_in'ADDRESS,def_seg,
                           def_off,r1_def_size );

--       advance the process that executes the command

            advance(ch_ch_user_synchro(active_user).seg_stack,
                                  success );

        END IF;

          active_user := 0;
          index := 1;

--      determine if while CCP was busy executing the previous
--      commands, some user handleror active user sent
--      a message
--
--      USER HANDLER
--      determine the user that sent the message comparing the
--      event count value, different value means that user was

          while index <= NUMBER_OF_USERS   LOOP
             read_evc(ch_param(index).seg_number_data,
                     evc_active, success );
             if (evc_active >
                     ch_param(index).evn_count_data) then
                  CCP_BUSY := TRUE;
                  active_user := index;
                  index := NUMBER_OF_USERS + 1;
             else
                  index := index + 1;
             END IF;
          END LOOP;

--       ACTIVE USER
--      determine the user that sent the message comparing the
--      event count value, different value means that user was

         index := 1;
         if active_user = 0 then
           while index <= NUMBER_OF_USERS  LOOP
               if ch_ch_user_synchro(index).active then
                 read_evc(ch_ch_user_synchro(index).seg_data,
                          ch_ch_evc_val, success);
                 if ch_ch_evc_val >
                      ch_ch_user_synchro(index).evc_data then
                      CCP_BUSY := TRUE;
                      index := NUMBER_OF_USERS + 1;
                 END IF;
               END IF;
```

```
            index := index + 1;
          END LOOP;
        END if;

      end LOOP;

--
--    ACTIVATE BDOS AGAIN AND WAIT UNTIL CHILD DELETE ITSEF
--

          ch_out.command := "bye      ";
          def_seg := lib_mk_sel(ldt_table,
                      ch_param(PREDOS).seg_number_data);
          def_off := 0;
          r1_def_size := comand_line'SIZE/8;
          move_bytes(get_ss(),ch_out'ADDRESS,def_seg,def_off,
                  r1_def_size);
          advance(ch_param(PREDOS).seg_number_stack,
                  success );
          read_evc(synchr_seg, evc_value, success);
          await(synchr_seg, evc_value+1, success );

--    ***************************************************************

--    ***************************************************************
--
--    BEGIN DELETING PROCESESS MODULE
--
--     This module eliminates the user's handler processes created
--     to support the active user processes, since each
--     process required of a specific number of segments that
--     were created in the CREATE_PROCESS module, these will
--     be terminated and deleted in this step

      proces := 0;
      while proces < NUMBER_OF_PROCESS  LOOP

        child_delete(proces, success );
        put_succ( "child deleted  ", success, w_class );
        put_ln( STDIO_W, w_class, "" );

        terminate_segment(ch_param(proces+1).seg_number_stack,
                           success );
        terminate_segment(ch_param(proces+1).seg_number_data,
                           success );
        terminate_segment(ch_param(proces+1).seg_number_code,
                           success );
        delete_segment(ch_param(proces+1).mentor_stack,
                  proces+1,success);  -- delete entries
        delete_segment(ch_param(proces+1).mentor_data,
                  proces+5,success);  -- delete entries data
```

```
    delete_segment(ch_param(proces+1).mentor_code,
                   proces+6,success);  -- delete entries code
    proces := proces + 1;

  end LOOP;

--  ***********************************************************

--  ***********************************************************
--

--     TERMINATE SEGMENTS AND DELETE SEGMENT MODULES
--
--     These modules will terminate and delete the segments
--     created previously to hold the mentor segment used to
--     create the user's stack segments and the user's data
--     segments, and the synchronization segment to
--     establish communication among processes.
--

--  terminate and delete synchronization segment

     terminate_segment(51,success);        --  segment 51
     delete_segment(31,11,success);        --  entry 11

--  terminate and delete mentor segment

    terminate_segment(31,success);         --  segment 31
    delete_segment(init.initial_seg(2), 5, success );


--  ***********************************************************

--  ***********************************************************
--
--    END PROCESS

  put_ln( STDIO_W, w_class, "  ***    GOOD BYE   ***" );

-- Infinite loop to prevent trap.  Could also await an
--                                 eventcount.
  success := 0;
  while success = 0 LOOP
        success := 2;
  END  loop;

END prmain;
```

APPENDIX B - USER HANDLER APPLICATION PROGRAM

This appendix lists the User Handler Application Program code. Only one program is provided since the three different programs. one for each different user, differ only in the port used on the RS-232 board. which is indicated below :

Port 6	User 1
Port 5	User 2
Port 3	User 3

The programs are called PUSER1. PUSER2. and PUSER3. The program listing for user 1 is detailed next.

```
pragma rangecheck(off);pragma debug(off);pragma
arithcheck(off); pragma enumtab(off);

WITH agate, agatej, arl, alit, alitj, strlit, files, util,
     proce;
PACKAGE BODY puser1 IS
USE agate, agatej, arl, alit, alitj, strlit, files, util,
    proce;

--    ************************************************************************

--     Constants used by the program
--
--    STDIO_W    -->  assigns logical device 1 to write
--    STDIO_R    -->  assigns logical device Ø to read
--    IO_PORT    -->  assigns the ports according with the
--                    follow detail :
--                              puser1    -->    port 6
--                              puser2    -->    port 5
--                              puser2    -->    port 3
--    PROCESS    -->  assigns prcess number 1,2,3 for puser1,
--                    puser2 and puser3 respectively


STDIO_W    : CONSTANT integer := 1;
STDIO_R    : CONSTANT integer := Ø;
IO_PORT    : CONSTANT integer := 6;
PROCESS    : CONSTANT integer := 1;
MEMORY_AVAILABLE : CONSTANT integer := 6Ø;
SEGMENTS_AVAILABLE : CONSTANT integer := 1ØØ;
NUMBER_PROCESSES : CONSTANT integer := 3;

--
--  Variables used by the program

w_class : access_class;
success : integer;
ch_in : input_message;
data_def_size : integer;
def_off        : integer;
def_seg        : integer;
ch_evc_val     : integer;
evc_ch_val     : integer;
rd_str         : string;
username       : string(8);
password       : string(8);
ch_class       : access_class;
ch_resource    : child_resource;
eco            : boolean;
end_prog       : boolean;
ch_access_level : level_record;
```

```
ch_level       : user_level;
entryx         : integer;
mentor         : integer;
seg_mode       : seg_access_type;
seg_number     : integer;
ch_parameters  : r1_parameters;
init : r1_process_def;

--   MAIN

Begin

   init := get_r1_def();      --  this sentence is obligatory

--   ******************************************************************

--  ATTACH TERMINAL MODULE
--
--  This module attaches port X to its process in order to
--  use it in I/O operations
--   attach terminal as write device

   attach_tew( IO_PORT, STDIO_W);

   w_class := init.resources.max_class;.

--  attach terminal as a read device

   attach_ter(IO_PORT, STDIO_R );

--  indicates that was activated ok

   put_ln(STDIO_W,w_class,"U S E R ");

--  synchronize with the main application (CCP) to indicate
--  that it was created ok and allow continued execution
--  using the await operator. It means that the process will
--  wait until the main application returns control to this
--  process


   advance(init.initial_seg(2), success );
   read_evc(init.initial_seg(0), evc_ch_val, success );
   await(init.initial_seg(0), evc_ch_val+1, success );

--   ******************************************************************

--   ******************************************************************
--
--  LOAD PARAMETERS MODULE
--
```

```
--  This module assigns parameters to the process that it
--  will create. These parameters are mentor number, entry
--  number, and segment number, used by stack, code and data
--  segments needed by the process to be created
--  (ACTIVE USER)

   load_param1(init,ch_parameters);
   load_access_class(init,ch_level);

   end_prog := false;
   while not end_prog LOOP

--  ****************************************************************
--
--  LOOP MODULE
--
--  This module executes several operations until the opera-
--  tor enters the word "bye". This means that the users work
--  is finished and termiantes this process execution. The
--  steps considered are :
--
--    a.- Clearance identification
--                  USERNAME and PASSWORD (login process)
--
--    b.- Create and makeknown mentor and synchronization
--        segments
--
--    c.- Load the child's resources (this will be
--        subtracted from the parent resources)
--
--    d.- Create child process (ACTIVE USER) single level
--
--    e.- Detach the device used for I/O, it will be used
--        by child
--
--    f.- Synchronize with the child created to indicates
--        what was created (USER CHILD)
--
--    g.- Synchronize with main application program(CCP) to
--        indicate that an ACTIVE USER was created. In turn
--        CCP will makeknown the segments that are synchro-
--        nized with ACTIVE USER (45,46 for user1;
--        47,48 for user2, 49,50 for user3)
--
--    h.- Synchronize main with active user and wait until
--        the active user finishes his job
--
--    i.- When ACTIVE USER terminates his job the user
--        handler recovers the resources assigned to his
--        child and terminates and deletes the segments
--        created to be mentor and synchronization, plus all
```

```
--          the segments needed to create the child
--          (loaded in step b)
--
--        j.- Synchronize with CCP to indicate that ACTIVE USER
--          is not longer active
--     END LOOP
--     clearance identification
--
       tlk_scr(STDIO_W,w_class,"");
       put_str(STDIO_W,w_class,"USERNAME ");
       eco := true;
       username := "";
       username := get_input(eco,w_class);
       put_ln(STDIO_W,w_class,"");    -- put cursor in next line
       if username = "bye" then
           end_prog := TRUE;
       else
           put_str(STDIO_W,w_class,"PASSWORD ");
           eco := false;
           password := "";
           password := get_input(eco,w_class);
           put_ln(STDIO_W,w_class,"");

           lock_for_level(username,password,ch_level,ch_class);

--     create mentor to the child process

           ch_access_level := ch_level(4);  -- min access class
           mentor := init.initial_seg(1);
           entryx := 3;
           ch_class := ch_access_level.min;
--     create the mentor segment
           cr_segment(init,mentor,entryx,ch_class,success);
           if success /= 0 then
              put_succ("success value 04 is ",success,w_class);
              put_ln(STDIO_W,w_class,"");
           END IF;

--     makeknown this segment

           seg_mode := r_w;
           seg_number := 31;
           mk_segment(init,mentor,entryx,seg_number,seg_mode,
                                         success);
           if success /= 0 then
              put_succ("success value 04 is ",success,w_class);
              put_ln(STDIO_W,w_class,"");
           END IF;

           ch_access_level.min := ch_access_level.max;
-- single level
```

```
--      load the resources that the child will have
--
--          memory   --> 60  ( format B24)
--          segments --> 100
--          processes -> 2;

        ch_resource.memory := MEMORY_AVAILABLE;
        ch_resource.segments := SEGMENTS_AVAILABLE;
        ch_resource.processes := NUMBER_PROCESSES;



--   SYNCHRONIZATION : code segment will be used to
--                     synchr. parent and its child, but
--                     initial_seg(2) is used to synchr.
--                     child with main application program

         cr_process(init,ch_parameters,ch_access_level,
          process,ch_resource,init.initial_seg(2),success);
         if success /= 0 then
            put_succ("success value 06 is ",success,w_class);
            put_ln(STDIO_W,w_class,"");
         END IF;

         read_evc(ch_parameters.seg_number_code,
                  ch_evc_val,success );

--
         detach_device(STDIO_W, success);
         detach_device(STDIO_R, success);
         await(ch_parameters.seg_number_code,ch_evc_val+1,
                                             success);

--   synchronize with main application program to create
--   segments to hold stack and data (will be used as synchr.
--   segments with the new process)


         def_seg := lit_mk_sel(ldt_table,init.initial_seg(3));
         def_off := 0;
         ch_in.input_one := username;
         ch_in.input_result := "";
         ch_in.input_class := w_class;
         data_def_size   := (input_message'SIZE/8);
         move_bytes(get_ss(), ch_in'ADDRESS, def_seg, def_off,
                    data_def_size);

--      synchronization process

         advance(init.initial_seg(3), success );
         advance(init.initial_seg(2), success );
```

```
        read_evc(init.initial_seg(0), evc_ch_val, success );

        await( init.initial_seg(0), evc_ch_val+1, success );

        read_evc(ch_parameters.seg_number_code,
                    ch_evc_val,success );

        advance(ch_parameters.seg_number_stack,success);
--
--    will await until child self delete
--
        await(ch_parameters.seg_number_code,ch_evc_val+1,
                              success);

        if success /= 0 then
           put_succ("success value 10 is ",success,w_class);
           put_ln(STDIO_W,w_class,"");
        END  IF;

--      attach I/O terminals again and delete segments created
--      created for the child process

        attach_tew(IO_PORT,STDIO_W);
        attach_ter(IO_PORT,STDIO_R);

--   delete segments

        terminate_segment(ch_parameters.seg_number_stack,
                             success);
        terminate_segment(ch_parameters.seg_number_code,
                             success);
        terminate_segment(ch_parameters.seg_number_data,
                             success);

        delete_segment(31,ch_parameters.entry_stack,success);
        delete_segment(31,ch_parameters.entry_data,success);

        terminate_segment(31, success);  -- terminate mentor
        delete_segment(init.initial_seg(1),3,success);
        child_delete(PROCESS-1,success);

--   communicate with main application program to delete the
--   segments to hold stack and data that were created to
--   synchronize main with child


        def_seg := lit_mk_sel(ldt_table,init.initial_seg(3));
        def_off := 0;
        ch_in.input_one := username;
        ch_in.input_result := "";
        ch_in.input_class := w_class;
```

```
          data_def_size    := (input_message'SIZE/8);
          move_bytes(get_ss(), ch_in'ADDRESS,def_seg, def_off,
                            data_def_size);

--     synchronization process , the control will return to
--     main application program CCP

          advance(init.initial_seg(3), success );
          advance(init.initial_seg(2), success );
          read_evc(init.initial_seg(0), evc_ch_val, success );
          await( init.initial_seg(0), evc_ch_val+1, success );

       end if;

end LOOP;

--   synchronize with main application program to tell that
--   the user no longer will use the terminal

          def_seg := lib_mk_sel(ldt_table,init.initial_seg(3));
          def_off := 0;
          ch_in.input_one := username;
          ch_in.input_result := "";
          ch_in.input_class := w_class;
          data_def_size    := (input_message'SIZE/8);
          move_bytes(get_ss(), ch_in'ADDRESS,def_seg, def_off,
                            data_def_size);

    detach_device( STDIO_R, success);
    detach_device( STDIO_W, success );
    advance(init.initial_seg(3), success );
    self_delete( init.initial_seg( 2 ), success );
    if success /= 0 then
       attach_tew(IO_PORT, STDIO_W );
       put_succ("successor is ",success, w_class);
    END if;

END ruser1;
```

APPENDIX C - ACTIVE USER APPLICATION PROGRAM

This appendix lists the Active User Application Program code. Only one program is provided since the three different programs, one for each different user, differ only in the port used on the RS-232 board, which is indicated below :

Port 6	User 1
Port 5	User 2
Port 3	User 3

The programs are called PRCHL1, PRCHL2, and PRCHL3. The program listing for user 1 is detailed next.

```
pragma rangecheck(off);pragma debug(off);
pragma arithcheck(off);pragma enumtab(off);

WITH agate, agatej, arl, alib, alibj, strlib, files, util;
PACKAGE BODY prchl1 IS
USE agate, agatej, arl, alib, alibj, strlib, files, util;

--  ****************************************************

--  Constants used by the program
--
--   STDIO_W    -->   assigns logical device 1 to write
--   STDIO_R    -->   assigns logical device 2 to read
--   IO_PORT    -->   assigns the ports according with the
--                    following detail :
--                          puser1   -->  port 6
--                          puser2   -->  port 5
--                          puser3   -->  port 3


STDIO_W   : CONSTANT integer := 1;
STDIO_R   : CONSTANT integer := 0;
IO_PORT   : CONSTANT integer := 6;


--
--  variables used by the program

w_class : access_class;
success : integer;
ch_in : input_message;
data_def_size : integer;
def_off       : integer;
def_seg       : integer;
evc_ch_val    : integer;
rd_str        : string;
eoo           : boolean;
end_prog      : boolean;
init : rl_process_def;

--  MAIN

Begin

   init := get_rl_def();     --  this sentence is obligatory

--  ****************************************************

--  ATTACH TERMINAL MODULE
--
--  This module attaches port X to this process in order to
```

```
--  use it in I/O operations, the device will have the same
--  clearance that the process has

   w_class := init.resources.max_class;

-- attach terminals (read and write)

   attach_ter( IO_PORT, STDIO_R );

   attach_tew( IO_PORT, STDIO_W );

   put_ln(STDIO_W,w_class,"USER CHILD ");

--  Syncronize with USER HANDLER to indicate that it was
--  created ok and let him continue his execution using the
--  advance, and await operator (advance User Handler
--  eventcount to continue, and await to stop its process
--  and returns the control

   advance(init.initial_seg(1),success);
   read_evc(init.initial_seg(2),evc_ch_val,success);
   await(init.initial_seg(2),evc_ch_val+1,success);

--  ************************************************************

--  ************************************************************

--  LOOP MODULE
--
--  This module executes several operations until the
--  operator enters the word "bye" that means work is done
--  The steps considered are :
--
--      a.- Put prompt ("*)") indicating that is ready to
--          accept ACTIVE USER input messages
--
--      b.- Get the user's input message
--
--      c.- Load input message entered by the user into
--          segment data used to pass the information
--
--      d.- Synchronize with main application program CCP
--          waiting for the answer to the message sent
--
--      e.- Display the result of the message after it was
--          processed by CCP
--
--  END LOOP

   eco := TRUE;
   end_prog := false;
```

```
while not end_prog LOOP

--  get input messages from the terminal

    rd_str := "";


    put_str(STDIO_W,w_class,"*) ");
    rd_str := get_input(eco,w_class);
    put_ln( STDIO_W, w_class, "");    --  put the cursor in
                                      --  the next line

    def_seg := lit_mk_sel(ldt_table, init.initial_seg(3));
    def_off := 0;
    ch_in.input_one := rd_str;
    ch_in.input_result := "";
    ch_in.input_class := w_class;
    data_def_size   := (input_message'SIZE/8);
    move_bytes(get_ss(), ch_in'ADDRESS, def_seg, def_off,
               data_def_size);
    if ((rd_str = "bye")  or (success /= 0)) then
       end_prog := true;
    else

--   begin the synchronization process


       advance(init.initial_seg(3), success );
       advance(init.initial_seg(2), success );

       read_evc(init.initial_seg(0), evc_ch_val, success );

       await( init.initial_seg(0), evc_ch_val+1, success );

--  display the answer's message that was transmitted by CCP

       def_seg := lib_mk_sel(ldt_table,init.initial_seg(3));
       def_off := 0;
       data_def_size   := (input_message'SIZE/8);
       move_bytes(def_seg, def_off,get_ss(), ch_in'ADDRESS,
                  data_def_size);
       put_ln( STDIO_W, w_class, ch_in.input_result );

    end if;

end LOOP;

--   *******************************************************************
```

```
--  ************************************************************************
--
--   DETACH TERMINAL MODULE
--
--   This module returns the device's control to the user
--   handler
   detach_device( STDIO_R, success );
   detach_device( STDIO_W, success );

--  ************************************************************************

--  ************************************************************************
--
--   SELF DELETE MODULE
--
--   This module terminates the child process (ACTIVE USER)
--   and advances the eventcount of the segment indicated.
--   In this case it is the segment used to synchronize with
--   his parent (user handler)

   self_delete( init.initial_seg( 1 ), success );
   if success /= 0 then
      attach_tew(IO_PORT, STDIO_W );
      put_succ("successor is ",success, w_class);
   END if;

END prchl1;
```

APPENDIX D - PRBDOS APPLICATION PROGRAM

This appendix presents the application program used to simulate the behavior of the BDOS operating system. Because of time constraints this program only has code that shows the process that should be performed when BDOS in invoked, simulating the result in order to pass it to the CCP process. This program is called PRBDOS, and its listing is next.

```
pragma rangecheck(off); pragma debug(off);
pragma arithcheck(off);pragma enumtab(off);

WITH agate, agatej, ar1, alib, alibj, strlib, files, util;
PACKAGE BODY prtdos IS
USE agate, agatej, ar1, alib, alibj, strlib, files, util;

--   ***********************************************************
--
--   This program only simulates the behavior of the BDOS
--   work, in order to handle "disk files"
--
--   ***********************************************************

--   constants
STDIO_W    : CONSTANT integer := 1;
STDIO_R    : CONSTANT integer := 0;
IO_PORT    : CONSTANT integer := 3;


--
-- MAIN
w_class : access_class;
success : integer;
data_def_size : integer;
def_off          : integer;
def_seg          : integer;
evc_ch_val       : integer;
ch_comm : comand_line;
end_prog         : boolean;
--file_data         : files_data;
--seg_head         : segment_header;
--seg_data         : segment_data;
init : r1_process_def;

Begin
   init := get_r1_def();

--   ***********************************************************
--
--    attach terminal as write device
--
   w_class := init.resources.max_class;
-- attach_tew( IO_PORT, STDIO_W);
--
--
--   ***********************************************************

--         initialize directory
--
--   put_ln(STDIO_W, w_class, "B D O S     ",;
```

```
--
--   ***********************************************************
--
--      Synchronization with CCP to tell that it was created
--      without trouble
--
--     begin the synchronization process

           advance(init.initial_seg(2), success );

           read_evc(init.initial_seg(0),evc_ch_val,success );

           await( init.initial_seg(0),evc_ch_val+1,success );

--   PROGRAM WAITS UNTIL THE CONTROL IS PASSED FROM CCP
--
--   ***********************************************************

  end_prog := false;

--   ***********************************************************
--
--   Begin loop until CCP sends "bye"
--
  while not end_prog LOOP

--     get command line passed by CCP

       def_seg    := lib_mk_sel(ldt_table,init.initial_seg(3));
       def_off    := 0;
       data_def_size     := (comand_line'SIZE/8);
       move_bytes(def_seg,def_off,get_ss(),ch_comm'ADDRESS,
                  data_def_size);

--     put_ln(STDIO_W,w_class, ch_comm.command);

       if (ch_comm.command /= "bye      ") then

           IF ch_comm.command = "create   " THEN
              ch_comm.result := "file created";

--            create_file();

           ELSIF ch_comm.command = "delete   " then
              ch_comm.result := "file deleted";

--            delete_file();

           ELSIF ch_comm.command = "rename   " then
              ch_comm.result := "file renamed";
```

```
--          rename_file();

         ELSE
            ch_comm.result := "mess. processed ";
         END IF;


--
--
--   load the result to pass it to the parent process

       def_seg := lib_mk_sel(ldt_table,init.initial_seg(3));
       def_off := 0;
       data_def_size   := (comand_line'SIZE/8);
       move_bytes(get_ss(),ch_comm'ADDRESS,def_seg,def_off,
                  data_def_size);

--   ******************************************************************
--
--   Synchronize process with CCP in order to pass the
--   result

        advance(init.initial_seg(2), success );
        read_evc(init.initial_seg(0), evc_ch_val, success );
        await(init.initial_seg(0), evc_ch_val+1, success );

--
      else
        end_prog := true;

    end IF;

  end LOOP;

--   ******************************************************************
--
--   End of the program and self deletion process

-- detach_device( STDIO_R, success);
-- detach_device( STDIO_W, success );

   self_delete( init.initial_seg( 2 ), success );
   if success /= 0 then
     attach_tew(IO_PORT, STDIO_W);
     put_succ("SUCCESSOR IS   ",success,w_class);
   END if;

END prtdcs;
```

APPENDIX E - COMMON PROCEDURES UTILITY

This appendix contains the procedures and functions used to provide information when the system primitives are used. These were obtained from the demonstration program provided by Gemini Computers Inc., and modified to reflect a generic use by the application programs developed in this research. The programs are "PROCE.LIB" (contains the specifications) and "PROCE.PKG" (contains the code developed).

```
WITH agate, agatej, ar1, alib, alibj, strlib, util, files;
PACKAGE BODY proce IS
USE agate, agatej, ar1, alib, alibj, strlib, util, files;

-- Constants for device slots.

STDIO_W : CONSTANT integer := 1;
STDIO_R : CONSTANT integer := 0;
IO_PORT : CONSTANT integer := 0;  -- port 0 for main

-- Constants for  segments.

SIZE_MENTOR      : CONSTANT integer := 1; -- size mentor
                                          -- synchr. segmt

--   ****************************************************************
--
--   PROCEDURE CR_SEGMENT
--
--   This procedure completes the parameters needed by the
--   primitive create_segment. The record structure is
--   described in the file "agate.lib" provided by Gemini
--   Computers Inc.
--
--   The parameters received by this procedure are :
--     init         -->  initial process definition
--     mentor       -->  indicates the segment number that
--                       will be parent of this new segment
--     entryx       -->  indicates which entry number of the
--                       mentor is used to create this segment
--     class        -->  indicates the security level of the
--                       segment to be created
--     success      -->  output variable that indicates the
--                       result of the operation after call
--                       the primitive
--
--   This procedure is used to create the mentor and
--   synchronization segments
--
--
PROCEDURE cr_segment( init : in r1_process_def;
                      mentor  : in integer;
                      entrx   : in integer;
                      class   : in access_class;
                      success : out integer ) IS


cr_seg_str : create_seg_struct;
w_class : access_class;

BEGIN
```

```
    w_class := init.resources.min_class;
    cr_seg_str.mentor := mentor;
    cr_seg_str.entryx := entrx;
    cr_seg_str.limit := SIZE_MENTOR;
    cr_seg_str.class := class;

    create_segment( cr_seg_str, success );
    if ( success = 817 ) THEN
        put_ln( STDIO_W, w_class, "817 means segment already
                                    exists." );
    END if;
END cr_segment;

--  ***************************************************

--  ***************************************************

--  PROCEDURE DL_SEGMENT
--
--  This procedure completes the parameters needed by the
--  primitive delete_segment. The record structure is
--  described in the file "agate.lit" provided by Gemini
--  Computers Inc.
--
--  The parameters received by this procedure are :
--    init          -->  initial process definition
--    mentor        -->  indicates the segment number that
--                       will be parent of this new segment
--    entryx        -->  indicates which entry number of the
--                       mentor is used to create this segment
--    class         -->  indicates the security level of the
--                       segment to be created
--    success       -->  output variable that indicates the
--                       result of the operation after call
--                       the primitive
--

PROCEDURE dl_segment ( init : in rl_process_def;
                       mentorx : in integer;
                       seg_number : in integer;
                       success : out integer ) IS

mentor, entryx : integer;
w_class : access_class;

BEGIN
    w_class := init.resources.min_class;
    mentor := mentorx;
    entryx := seg_number;
    delete_segment( mentor, entryx, success );
END dl_seg_tst;
```

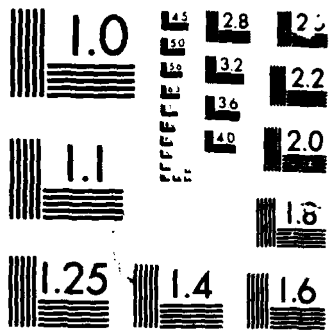
```
-- *****************************************************************

-- *****************************************************************
--
-- PROCEDURE MK_SEGMENT
--
-- This procedure completes the parameters needed by the
-- primitive makeknown_segment. The record structure is
-- described in the file "agate.lit" provided by Gemini
-- Computers Inc.
--
-- The parameters received by this procedure are :
--     init          --> initial process definition
--     mentor        --> indicates the segment number that
--                       will be parent of this new segment
--     entryx        --> indicates which entry number of the
--                       mentor is used to create this segment
--     number        --> indicates the number that the segment
--                       will have in the LDT
--     mode          --> indicates the kind of segment that
--                       will be created (r_w, r_e, etc.)
--     success       --> output variable that indicates the
--                       result of the operation after call
--                       the primitive
--
-- This procedure is used to makeknown the mentor and
-- synchronization segments
--
--

PROCEDURE mk_segment ( init : in r1_process_def;
                       mentor  : in integer;
                       entrx   : in integer;
                       number  : in integer;
                       mode    : in seg_access_type;
                       success : out integer ) IS

seg_rec : mk_kn_struct;
seg_ret_rec : mk_kn_return;
w_class : access_class;

BEGIN
    w_class := init.resources.min_class;
    seg_rec.mentor := mentor;
    seg_rec.entryx := entrx;
    seg_rec.seg_number := number;
    seg_rec.seg_mode := mode;
    seg_rec.prot_level := byte( 1 );      --ring 1 protection
    seg_rec.gate_number := NULL_INDEX;          -- no gate
```

```
    seg_rec.gate_prot := byte( 2 );
    makeknown_segment ( seg_rec, seg_ret_rec, success );
END mk_segment;

--  ***********************************************************

--  ***********************************************************
--
--  PROCEDURE MAKE_KNOWN_SYNC
--
--  This procedure effects several actions related to the
--  creation of special segments to synchronize the main
--  application program with the active user. Since the
--  segments were created by the active user this procedure
--  will only makeknown those in its own LDT table, and
--  swapin these in memory

--  The parameters received by this procedure are :
--     init          -->  initial process definition
--     ch_para       -->  indicates the parameters used to
--                        create the user handler process
--     ch_class      -->  indicates the access_class of the
--                        segment to be created
--     ch_active     -->  indicates which active user is
--                        trying to communicate with
--     ch_user_sync-->    is an output record that contains
--                        the segment numbers assigned to the
--                        synchronization segments
--     success       -->  output variable that indicates the
--                        result of the operation after call
--                        the primitive
--
--  This procedure is used to makeknown the synchronization
--  segments (main application - active user)
--
--
PROCEDURE make_know_sync( init : in rl_process_def;
                          ch_para : in rl_parameters;
                          ch_class : in access_class;
                          ch_active : in integer;
                          ch_user_sync : out users_active;
                          success : out integer ) IS

mentor : integer;
entryx : integer;
seg_mode : seg_access_type;
seg_number : integer;
class     : access_class;

BEGIN
```

```
--  make known root segment (code segment of each child)

    mentor := ch_para.seg_number_code;
    entryx := 3;
    seg_number := ch_para.synchr_chld_mentor;
    seg_mode := r_w;
    class := init.resources.min_class;
    mk_segment(init,mentor,entryx,seg_number,
                                        seg_mode,success);
    if success /= 0 then
       put_succ("success value 226 is ",success,ch_class);
       put_ln(STDIO_W,ch_class,"");
    end if;

--  make known stack segment

    mentor := ch_para.synchr_chld_mentor;
    entryx := 1;
    seg_number := ch_para.synchr_chld_stack;
    seg_mode := r_w;
    mk_segment(init,mentor,entryx,seg_number,
                                        seg_mode,success);
    if success /= 0 then
       put_succ("success value 227 is ",success,ch_class);
       put_ln(STDIO_W,ch_class,"");
    end if;

--  make known data  segment

    mentor := ch_para.synchr_chld_mentor;
    entryx := 2;
    seg_number := ch_para.synchr_chld_data;
    seg_mode := r_w;
    mk_segment(init,mentor,entryx,seg_number,
                                        seg_mode,success);
    if success /= 0 then
       put_succ("success value 228 is ",success,ch_cl[illegible]
       put_ln(STDIO_W,ch_class,"");
    end if;

    ch_user_sync(ch_active).seg_data :=
                                ch_para.synchr[illegible]
    ch_user_sync(ch_active).seg_stack :=
                                ch_para.synchr[illegible]

    swapin_segment(ch_user_sync(ch_a[illegible]
                                succ[illegible]
--  read event counts of each segment

    read_evc(ch_user_sync(ch_act[illegible]
             ch_user_sync(ch_acti[illegible]
```

AD-A175 908 DYNAMIC SHARING OF THE SYSTEM RESOURCES IN MULTILEVEL SECURE SYSTEM(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA M A REYES 26 SEP 86 2/2

UNCLASSIFIED F/G 9/2 NL


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

```
    terminate_segment(44,success);
    if success /= 0 then
       put_succ("success value @12 is ",success,ch_class);
       put_ln(STDIO_W,ch_class,"");
    end if;

END make_know_sync;


--  ***********************************************************


--  ***********************************************************
--
--  PROCEDURE TERMINATE_SYNCHR_SEG
--
--  This procedure terminates the segments makeknown
--  previously with the object to synchronize the communica-
--  tion between an active user and the main appl. program

--  The parameters received by this procedure are :
--    init          -->  initial process definition
--    ch_para       -->  indicates the parameters used to
--                       create the user handler process
--    ch_class      -->  indicates the access_class of the
--                       segment to be created
--    success       -->  output variable that indicates the
--                       result of the operation after call
--                       the primitive
--
--  This procedure is used to delete the synchronization
--  segments created before (main - active user)
--
PROCEDURE terminate_synchr_seg( init : in r1_process_def;
                                ch_para : in r1_parameters;
                                ch_class : in access_class;
                                success  : out integer ) IS


--  terminates the segments created to synchronize main
--  application program with the process created by the
--  child process leaving available the segment numbers in
--  the LDT

BEGIN
   terminate_segment(ch_para.synchr_chld_data, success );

   terminate_segment(ch_para.synchr_chld_stack, success );

END terminate_synchr_seg;
```

```
--  ************************************************************

--  ************************************************************
--
--  PROCEDURE FILL_INIT
--
--  This procedure fills the process definition record with
--  the data provided in the initial process definition plus
--  the resorces that the parent will pass to his child and
--  the access class of this specefic process

--  The parameters received by this procedure are :
--    init         -->  initial process definition
--    ch_init      -->  output process definition
--    ch_resource  -->  indicates the resources passed by its
--                      parent
--    success      -->  output variable that indicates the
--                      result of the operation after call
--                      the primitive
--
--

PROCEDURE fill_init( init : in r1_process_def;
                     ch_init : out r1_process_def;
                     ch_resource : in child_resource;
                     ch_access : level_record ) IS

--     fill in the initial process record of a child
--     process called by crt_proc_tst.

BEGIN
    ch_init.cpu := init.cpu;
    ch_init.num_cpu := init.num_cpu;
    ch_init.num_kst := init.num_kst;
    ch_init.root_access := init.root_access;
    ch_init.s_seg := 3;
    ch_init.resources.priority :=
                init.resources.priority; --same as parent.
    b24_frm_integer( ch_resource.memory,
                              ch_init.resources.memory );
    ch_init.resources.processes := ch_resource.processes;
    ch_init.resources.segmnts := ch_resource.segments;

 --   this will be modified with the specific access class
 --   of each process

    ch_init.resources.min_class := ch_access.min;
    ch_init.resources.max_class := ch_access.max;
    ch_init.ring_num := byte( 1 );
```

```
    ch_init.sp2 := 0;
END fill_init;

--  ************************************************************

--  ************************************************************
--
--  PROCEDURE CR_PROCESS
--
--  This procedure performs all the operations necessary to
--  create a child process, this operations include
--  makeknown the code segment of the child, creation of
--  stack and data segments, fill the addess space
--  specification and process creation
--
--  The parameters received by this procedure are :
--    init        -->  initial process definition
--    ch_par      -->  parameters to create a child
--                     (segment numbers,entry numbers,etc)
--    process     -->  indicates the process number to be
--                     created (example active user 1 )
--    ch_resource -->  indicates the resources that the
--                     child will have
--    synchr_seg  -->  indicates the segment that is used
--                     to synchronize this new process with
--                     its parent
--    success     -->  output variable that indicates the
--                     result of the operation after call
--                     the primitive
--
--


PROCEDURE cr_process( init : in r1_process_def;
                       ch_par     : in r1_parameters;
                       ch_access : in level_record;
                       proces  : in integer;
                       ch_resource  : in child_resource;
                       synchr_seg : in integer;
                       success : out integer ) IS


chld_seg : r1_seg_struct;-- r1_addr_array for child segment
ch_init : r1_process_def;     -- r1_process_def for child
seg_rec : create_seg_struct; -- used to create stack segment
seg1_mkn : mk_kn_struct; -- used to make known stack segment
seg1_ret : mk_kn_return;
crt_rec : r1_cp_struct;       -- create process structure
ch_seg_list : seg_array;
```

```
ch_inpt_mess : input_message;
data_def_size : integer;
end_chld       : boolean;
w_class : access_class;
evc_value : integer;
stack_size : integer;
seg_mgr_bytes : integer;
def_off : integer;
def_seg : integer;
r1_def_size : integer;
dummy : integer;

-- constants for determining stack size

r1_stack_size : CONSTANT integer := 16#AFF#;
vect_size : CONSTANT integer := 4;

BEGIN
    w_class := ch_access.min;

    seg1_mkn.mentor := ch_par.mentor_code; -- appl. root
    seg1_mkn.entryx := ch_par.entry_code;
    seg1_mkn.seg_number := ch_par.seg_number_code;
    seg1_mkn.seg_mode := r_e;
    seg1_mkn.prot_level := byte( 1 );
    seg1_mkn.gate_number := NULL_INDEX;        -- no gate
    makeknown_segment( seg1_mkn, seg1_ret, success );
    if success /= 0 then
       put_succ("success value is ",success,w_class);
       put_ln(STDIO_W,w_class,"");
    END IF;


--   address spec for child's stack

    chld_seg.seg_number := ch_par.seg_number_stack;
    chld_seg.seg_mode := r_w;
    chld_seg.swapin := TRUE;
    chld_seg.protect := byte( 1 );
    crt_rec.r1_addr_array( 0 ) := chld_seg;

--   address spec for child's code

    chld_seg.seg_number := ch_par.seg_number_code;
    chld_seg.seg_mode := r_e;
    chld_seg.swapin := TRUE;
    chld_seg.protect := byte( 1 );
    crt_rec.r1_addr_array( 1 ) := chld_seg;

--   address spec for child's mentor
```

```
    chld_seg.seg_number := synchr_seg;
    chld_seg.seg_mode := n_a;
    chld_seg.swapin := TRUE;
    chld_seg.protect := byte( 1 );
    crt_rec.r1_addr_array( 2 ) := chld_seg;

--   address spec for trap handler segment

    chld_seg.seg_number := init.initial_seg(4);
    chld_seg.seg_mode := r_e;
    chld_seg.swapin := TRUE;
    chld_seg.protect := byte( 1 );
    crt_rec.r1_addr_array( 4 ) := chld_seg;

--   address spec for child's data

    chld_seg.seg_number := ch_par.seg_number_data;
    chld_seg.seg_mode := r_w;
    chld_seg.swapin := TRUE;
    chld_seg.protect := byte( 1 );
    crt_rec.r1_addr_array( 3 ) := chld_seg;

--  fill the order in which the segments will be passed

    ch_seg_list(0)    :=  ch_par.seg_number_stack;
    ch_seg_list(1)    :=  ch_par.seg_number_code;
    ch_seg_list(2)    := synchr_seg;
    ch_seg_list(3)    := ch_par.seg_number_data;
    ch_seg_list(4)    := init.initial_seg(4);


--  calculate required stack size.
--  (in the future will calculate based on data in "CMD"
--  file header but now just use constant.)

    seg_mgr_bytes := ( stack_header'SIZE/8 ) +
        ( init.num_kst * ( kst_entry'SIZE/8 )) +
                ( kst_header'SIZE/8 );
    stack_size := r1_stack_size+vect_size+seg_mgr_bytes +
                        ( r1_process_def'SIZE/8 );

--     create and make known child's stack segment

    seg_rec.mentor := ch_par.mentor_stack;
    seg_rec.entryx := ch_par.entry_stack;
    seg_rec.limit := stack_size - 1;
    seg_rec.class := ch_access.max;  --  **********
    create_segment( seg_rec, success );
    if success /= 0 then
       put_succ("success value aa is ",success,w_class);
       put_ln(STDIO_W,w_class,"");
```

```
END IF;
seg1_mkn.mentor := ch_par.mentor_stack;
seg1_mkn.entryx := ch_par.entry_stack;
seg1_mkn.seg_number := ch_par.seg_number_stack;
seg1_mkn.seg_mode := r_w;
seg1_mkn.prot_level := byte( 1 );
seg1_mkn.gate_number := NULL_INDEX;
seg1_mkn.gate_prot   := byte( 0 );
makeknown_segment( seg1_mkn, seg1_ret, success );
if success /= 0 then
   put_succ("success value a is ",success,w_class);
   put_ln(STDIO_W,w_class,"");
END IF;
swapin_segment( ch_par.seg_number_stack, success );
if success /= 0 then
   put_succ("success value b is ",success,w_class);
   put_ln(STDIO_W,w_class,"");
END IF;

--    create and make known child's data segment

seg_rec.mentor := ch_par.mentor_data;
seg_rec.entryx := ch_par.entry_data;
seg_rec.limit := input_message'SIZE/8;
seg_rec.class := ch_access.max;  --  ***********
create_segment( seg_rec, success );
if success /= 0 then
   put_succ("success value cc is ",success,w_class);
   put_ln(STDIO_W,w_class,"");
END IF;
seg1_mkn.mentor := ch_par.mentor_data;
seg1_mkn.entryx := ch_par.entry_data;
seg1_mkn.seg_number := ch_par.seg_number_data;
seg1_mkn.seg_mode := r_w;
seg1_mkn.prot_level := byte( 1 );
seg1_mkn.gate_number := NULL_INDEX;
seg1_mkn.gate_prot   := byte( 0 );
makeknown_segment( seg1_mkn, seg1_ret, success );
if success /= 0 then
   put_succ("success value c is ",success,w_class);
   put_ln(STDIO_W,w_class,"");
END IF;
swapin_segment( ch_par.seg_number_data, success );
if success /= 0 then
   put_succ("success value d is ",success,w_class);
   put_ln(STDIO_W,w_class,"");
END IF;

--  fill in childs r1_process_def

fill_init( init, ch_init, ch_resource, ch_access );
```

```
-- determine segment & offset of r1_process_def initial
-- record

    def_seg := lit_mk_sel( ldt_table,
                           ch_par.seg_number_stack );
    def_off := stack_size - ( vect_size + seg_mgr_bytes +
               r1_process_def'SIZE/8 );

--  move ch_init into proper place in child's stack segment

    r1_def_size := ( r1_process_def'SIZE )/8;
    move_bytes( get_ss(), ch_init'address, def_seg, def_off,
                                      r1_def_size );

--  fill in remainder of create_process_structure

    crt_rec.ip := 128;  -- skip command file header (80 hex)
    crt_rec.spx := def_off;        -- set childs stack pointer
    crt_rec.sp1 := stack_size - (vect_size + seg_mgr_bytes);
    crt_rec.sp2 := 0;             -- no ring 2 stack
    crt_rec.vec_seg := 0;         -- r1 address array element 0
    crt_rec.vec_off := stack_size - vect_size;
    crt_rec.child_num := proces-1;
    crt_rec.priority := ch_init.resources.priority;
    crt_rec.memory := ch_init.resources.memory;
    crt_rec.processes := ch_init.resources.processes;
    crt_rec.segmnts := ch_init.resources.segmnts;
    crt_rec.min_class := ch_init.resources.min_class;
    crt_rec.max_class := ch_init.resources.max_class;

--  read event count so we know when child has self_deleted

--  read_evc(synchr_seg,evc_value, success );

--  create the process

    create_process( crt_rec, success );
    if success /= 0 THEN
        put_succ( " create process success = ",
                              success, w_class );
    END if;

END cr_process;

END proce;
```

```
WITH agate, agatej, ar1, alib, alibj, strlib, util, files;
PACKAGE  proce IS
USE agate, agatej, ar1, alib, alibj, strlib, util, files;

--   ***************************************************************

--   THIS PROGRAM IS PROCE.LIB
--
--   Contains the specifications needed by PROCE.PKG program
--
--   ***************************************************************

PROCEDURE cr_segment( init : in r1_process_def;
                      mentor : in integer;
                      entrx  : in integer;
                      class  : in access_class;
                      success : out integer ) ;


PROCEDURE dl_segment ( init : in r1_process_def; success :
                       out integer );


PROCEDURE mk_segment ( init : in r1_process_def;
                       mentor  : in integer;
                       entrx   : in integer;
                       number  : in integer;
                       mode    : in seg_access_type;
                       success : out integer );

PROCEDURE make_know_Sync( init : in r1_process_def;
                       ch_para : in r1_parameters;
                       ch_class : in access_class;
                       ch_active : in integer;
                       ch_user_sync : out users_active;
                       success    : out integer);

PROCEDURE terminate_synchr_seg(init : in r1_process_def;
                       ch_para : in r1_parameters;
                       ch_class : in access_class;
                       success    : out integer);

PROCEDURE fill_init( init : in r1_process_def;
                     ch_init : out r1_process_def;
                     ch_resource : in child_resource;
                     ch_access : in level_record ) ;

PROCEDURE cr_process( init : in r1_process_def;
                      ch_para  : in r1_parameters;
                      ch_access : in level_record;
```

```
                    proces  : in integer;
                    ch_resource  : in child_resource;
                    synchr_seg : in integer;
                    success : out integer);

END proce;
```

APPENDIX F - SERVICE ROUTINES AND ADDITIONAL DATA STRUCTURE

This appendix contains the procedures and functions used by the application programs related to execution of I/O operations. It also contains additional data structures necessary to run specific application programs (parameters, record's description, constants, etc.). The program is "Files" and is composed of two modules, "FILES.LIB" (contains the specifications of records, functions and procedures used) and "FILES.PKG" (contains the code developed for each procedure or function).

```
WITH agate, agatej, ar1, alit, alitj, strlit, util;
PACKAGE BODY files IS
USE agate, agatej, ar1, alit, alitj, strlit, util;

STDIO_W : CONSTANT integer := 1;
STDIO_R : CONSTANT integer := 2;


PROCEDURE b24_frm_integer( in_val : in integer;
                                      b24_val : out b24_type )IS

--    Routine to convert an integer into a
--    B24_type variable ( 3-bytes )

BEGIN
    b24_val.byte2 := byte( 0 );
    b24_val.byte1 := hi( in_val );
    b24_val.byte0 := lo( in_val );
END b24_frm_integer;


PROCEDURE put_ln ( ldev : in integer;
                   w_class : in access_class;
                   str : in string ) IS

--    put a string on device ldev with cr and lf

out_buf : string( 82 );
success : integer;
wt_sio : wt_seq_struct;
size_str : integer;
CR : CONSTANT integer := 13;
LF : CONSTANT integer := 10;

BEGIN
    out_buf := str;
    size_str := length( str );
    out_buf := out_buf & char_to_str( character'val( CR ));
    out_buf := out_buf & char_to_str( character'val( LF ));
    wt_sio.device := ldev;
    wt_sio.data_off := out_buf'ADDRESS + 1;
    wt_sio.data_seg := get_ss();
    wt_sio.count := size_str + 2;
    wt_sio.class := w_class;
    write_sequential( wt_sio, success );
END put_ln;


PROCEDURE blk_scr ( ldev : in integer;
                    w_class : in access_class;
                    str : in string ) IS
```

```
--    blank the screen and put the cursor in the
--    first position

out_buf : string( 82 );
success : integer;
wt_sio : wt_seq_struct;
size_str : integer;
ESC : CONSTANT integer := 27;
E   : CONSTANT integer := 45;

BEGIN
    out_buf := str;
    size_str := length( str );
    out_buf := out_buf & char_to_str( character'val( ESC ));
    out_buf := out_buf & char_to_str( character'val( E ));
    wt_sio.device := ldev;
    wt_sio.data_off := out_buf'ADDRESS + 1;
    wt_sio.data_seg := get_ss();
    wt_sio.count := size_str + 2;
    wt_sio.class := w_class;
    write_sequential( wt_sio, success );
END blk_scr;


PROCEDURE get_str ( ldev : in integer;
                    r_class : out access_class;
                    str : out string ) IS

--    get a string from device ldev.
in_buf  : string( 82 );
success : integer;
rd_sio : rd_seq_struct;
rd_ret : rd_seq_return;
size_str : integer;

BEGIN
    rd_sio.data_off := in_buf'ADDRESS + 1;
    rd_sio.device := ldev;
    rd_sio.data_seg := get_ss();
    read_sequential( rd_sio, rd_ret, success );
    in_buf( 0 ) := character'val( rd_ret.count );
    str := in_buf;
    r_class := rd_ret.class;
END get_str;


PROCEDURE put_str ( ldev : in integer;
                    w_class : in access_class;
                    str : in string ) IS
```

```
--   put a string on device ldev.

out_buf : string;
success : integer;
wt_sio : wt_seq_struct;
size_str : integer;

BEGIN
    out_buf := str;
    size_str := length( str );
    wt_sio.device := ldev;
    wt_sio.data_off := out_buf'ADDRESS + 1;
    wt_sio.data_seg := get_ss();
    wt_sio.count := size_str;
    wt_sio.class := w_class;
    write_sequential( wt_sio, success );
END put_str;


PROCEDURE put_dec( ldev : in integer;
                   w_class : in access_class;
                   dval : in integer ) IS

--  put the string equivalent of a integer on the terminal
--  screen.

out_buf : string( 10 );

BEGIN
    out_buf := Int_to_str( dval );
    put_str( ldev, w_class, out_buf );
END put_dec;


PROCEDURE put_succ( in_str : in string;
                    dec_val : in integer;
                    w_class : in access_class ) IS

--   print a string and an integer on device attached in
--   slot STDIO_W (should be a serial terminal).

BEGIN
    put_str( STDIO_W, w_class, in_str );
    put_dec( STDIO_W, w_class, dec_val );
    put_ln( STDIO_W, w_class, "" );
END put_succ;
```

```
FUNCTION get_input( eco : in boolean;
                    rd_class : in access_class ) RETURN string IS

--  Gets an input string from the terminal and echoes the
--  input if the echo option is on. It also converts all the
--  input to lower case


--  constants
STDIO_R : CONSTANT integer := 0;
STDIO_W : CONSTANT integer := 1;

rd_str : string;
ind    : integer;
values : integer;
inp_ch : string(1);
w_class  : access_class;end_input : boolean;

BEGIN

    w_class   := rd_class;
    end_input := false;
    ind       := 1;
    rd_str := "";
    while not end_input LOOP
      get_str(STDIO_R, w_class, inp_ch);
      if inp_ch(1) in 'A'..'Z' then
        inp_ch(1) :=
                  character'val(character'pos(inp_ch(1))+32);
      end if;
      if ( character'pos(inp_ch(1)) = 13 )     then
           end_input := true;
      else
         if eco then
           put_str(STDIO_W, rd_class, inp_ch );
         END IF;
         rd_str := insert( inp_ch, rd_str, ind );
         ind := ind + 1;
      end if;
    end LOOP;
    RETURN rd_str;

END get_input;

PROCEDURE attach_tew( IO_PORT : in integer;
                      LDEV : in integer) IS

--   attach serial port for writing.
```

```
mode      : attach_struct;
w_class   : access_class;
success   : integer;

BEGIN

    mode.dev_name := siow;
    mode.siow_rec.dev_num := io_port;
    mode.siow_rec.dev_type := io;
    mode.siow_rec.dev_id := LDEV;
    mode.siow_rec.mr1 := byte( 16#24D# );
    mode.siow_rec.mr2 := byte( 16#03E# );
    mode.siow_rec.io_mode := asrt_rts;
    attach_device( mode, success );

END attach_tew;

PROCEDURE attach_ter( IO_PORT : in integer;
                      LDEV : in integer) IS

--   attach serial port for reading.

mode_r    : attach_struct;
w_class   : access_class;
success   : integer;

BEGIN

    mode_r.dev_name := sior;
    mode_r.sior_rec.dev_num := io_port;
    mode_r.sior_rec.dev_type := io;
    mode_r.sior_rec.dev_id := LDEV;
    mode_r.sior_rec.mr1 := byte( 16#24D# );
    mode_r.sior_rec.mr2 := byte( 16#03E# );
    mode_r.sior_rec.io_mode := asrt_dtr;
    mode_r.sior_rec.delim_active := FALSE;
    mode_r.sior_rec.delimiter := byte( 13 );
    mode_r.sior_rec.maximum := 1;
                  -- Only reads one character at a time.
    attach_device( mode_r, success );

END attach_ter;


PROCEDURE load_param(init : in r1_process_def;
                     ch_para : out r1_param) IS


--   Produces a table of parameters with information needed
```

```
--  by the main application program (segment number, entry,
--  mentor).

INITIAL   : CONSTANT integer := 31;
NEXT_NUMBER_FREE   : CONSTANT integer := 40;
CH_SYNCHR_MENTOR   : CONSTANT integer := 44;

index : integer;
next_segment : integer;
data_number : integer;
ch_param   : r1_parameters;
usr_level : level_record;
synchr_chld : integer;


BEGIN
  next_segment := INITIAL;
  data_number := NEXT_NUMBER_FREE;
  synchr_chld := CH_SYNCHR_MENTOR + 1;
                                      -- next segment available
  index          := 1;
  while index < 5 LOOP
    ch_param.entry_stack := index;
    ch_param.mentor_stack := INITIAL;
    next_segment := next_segment + 1;
    ch_param.seg_number_stack := next_segment;
    next_segment := next_segment + 1;
    ch_param.seg_number_code := next_segment;
    ch_param.entry_code := index + 6;
    ch_param.mentor_code := init.initial_seg(2);
    ch_param.entry_data := index + 4;
    ch_param.mentor_data := INITIAL;
    ch_param.seg_number_data := data_number;
    data_number := data_number + 1;
    if index < 4 then
       ch_param.synchr_chld_mentor := CH_SYNCHR_MENTOR;
       ch_param.synchr_chld_stack  := synchr_chld;
       synchr_chld := synchr_chld + 1;
       ch_param.synchr_chld_data   := synchr_chld;
       synchr_chld := synchr_chld + 1;
    else
       ch_param.synchr_chld_mentor := 0;
       ch_param.synchr_chld_stack   := 0;
       ch_param.synchr_chld_data    := 0;
    END IF;
    ch_para(index) := ch_param;
    index := index + 1;
  END LOOP;

END load_param;
```

```
PROCEDURE load_access_class(init : in r1_process_def;
                            usr_access : out user_level ) IS
--
--  Produces a table with the security access level depen-
--  ding on the user level

usr_level  : level_record;

BEGIN
    usr_level.min.compromise.int0 := 0;
    usr_level.min.compromise.int1 := 0;
    usr_level.min.integrity.int0  := 0;
    usr_level.min.integrity.int1  := 21504;
    usr_level.max.compromise.int0 := 6;
    usr_level.max.compromise.int1 := 0;
    usr_level.max.integrity.int0  := 0;
    usr_level.max.integrity.int1  := 21504;
    usr_access(TOP_SECRET) := usr_level;


    usr_level.min.compromise.int0     := 0;
    usr_level.min.compromise.int1     := 0;
    usr_level.min.integrity.int0      := 0;
    usr_level.min.integrity.int1      := 21504;
    usr_level.max.compromise.int0     := 4;
    usr_level.max.compromise.int1     := 0;
    usr_level.max.integrity.int0      := 0;
    usr_level.max.integrity.int1      := 21504;
    usr_access(SECRET) := usr_level;

    usr_level.min.compromise.int0 := 0;
    usr_level.min.compromise.int1 := 0;
    usr_level.min.integrity.int0  := 0;
    usr_level.min.integrity.int1  := 21504;
    usr_level.max.compromise.int0 := 2;
    usr_level.max.compromise.int1 := 0;
    usr_level.max.integrity.int0  := 0;
    usr_level.max.integrity.int1  := 21504;
    usr_access(CONFIDENTIAL) := usr_level;

    usr_level.min.compromise.int0 := 0;
    usr_level.min.compromise.int1 := 0;
    usr_level.min.integrity.int0  := 0;
    usr_level.min.integrity.int1  := 21504;
    usr_level.max.compromise.int0 := 0;
    usr_level.max.compromise.int1 := 0;
    usr_level.max.integrity.int0  := 0;
    usr_level.max.integrity.int1  := 21504;
    usr_access(UNCLASSIFIED) := usr_level;
```

```
END load_access_class;

PROCEDURE load_param1(init : in r1_process_def;
                      ch_param : out r1_parameters ) IS

--  Produces a table of parameters with information needed
--  by the User Handler


MENTOR       : CONSTANT integer := 31;

BEGIN
    ch_param.entry_stack := 1;                    -- always 1
    ch_param.mentor_stack := MENTOR;
    ch_param.seg_number_stack := 32;              --   "    32
    ch_param.seg_number_code := 33;
    ch_param.entry_code  := 4;
    ch_param.mentor_code := init.initial_seg(1);
    ch_param.entry_data := 2;
    ch_param.mentor_data := MENTOR;
    ch_param.seg_number_data := 34;

END load_param1;

PROCEDURE load_child_active
                         (usr_active : out users_active) IS

--  Initializes the Active User record to FALSE. It lets
--  CCP load the Active User segments each time a false
--  record is found


index : integer;

BEGIN
   index := 1;
   while index < 3 LOOP
     usr_active(index).active := FALSE;
     usr_active(index).seg_data := 0;
     usr_active(index).seg_stack := 0;
     usr_active(index).evc_data := 0;
     usr_active(index).evc_stack := 0;
     index := index + 1;
   end LOOP;

END load_child_active;

PROCEDURE look_for_level(username : in string;
                         password : in string;
```

```
                           ch_access : in user_level;
                           ch_class : out  access_class) IS

--  Simulates the Logon process, loading the access class of
--  the user depending on the Username and Pasword


BEGIN
     if password = "falcon1" then
        ch_class := ch_access(1).max;
     elsif password = "falcon" then
        ch_class := ch_access(2).max;
     ELSIF password = "secret" then
        ch_class := ch_access(3).max;
     ELSE
        ch_class := ch_access(4).max;
     END IF;

END lock_for_level;

PROCEDURE initialize_tables(seg_table : out files_data;
                           seg_head   : out segment_header) IS

--  Initializes the internal tables that will simulate the
--  automatic creation of segments numbers using the LDT
--  table


index        : integer;
w_class      : access_class;

BEGIN
   w_class.integrity.int0 := 0;
   w_class.integrity.int1 := 0;
   w_class.compromise.int1 := 0;
   w_class.compromise.int0 := 0;
   seg_head.max_files_stored    := 0;
   seg_head.next_avail_seg      := INITIAL_FREE_SEGMENT;
   seg_head.next_avail_ent      := INITIAL_FREE_ENTRY;
   seg_head.next_avail_men      := INITIAL_FREE_MENTOR;
   seg_head.max_open_seg        := INITIAL_FREE_SEGMENT;
   seg_head.max_open_ent        := INITIAL_FREE_ENTRY;
   seg_head.max_open_men        := INITIAL_FREE_MENTOR;

--  initialization of array that holds files information

   index := 0;
   while index < MAX_NUMBER_OF_FILES + 1 LOOP

     seg_table(index).number      := 0;
     seg_table(index).entrys      := 0;
```

```
      seg_table(index).mentor        := 0;
      seg_table(index).file_name     := "";
      seg_table(index).access_cla := w_class;
      seg_table(index).next_avail_seg := INITIAL_FREE_SEGMENT;
      seg_table(index).next_avail_ent := INITIAL_FREE_ENTRY;
      seg_table(index).next_avail_men := INITIAL_FREE_MENTOR;
      index := index + 1;

   end LOOP;

END initialize_tables;

FUNCTION check_if_exists_file_name
                (seg_table : in files_data;
                 file_name : in string) RETURN boolean IS

--  check if file name declared in input command exists or
--  does not

index     : integer;
answer    : boolean;

BEGIN
    index := 0;
    answer := FALSE;
    while index < MAX_NUMBER_OF_FILES + 1 LOOP

       if seg_table(index).file_name = file_name then
          answer  := TRUE;
          RETURN answer;
       else
          index := index + 1;
       end IF;

    END LOOP;
    RETURN answer;

END check_if_exists_file_name;

PROCEDURE convert(ch_in : in  input_message;
                  w_class  : in access_class;
                  ch_out   : out comand_line) IS

--  this procedure assembles the commad line using the input
--  message typed by the user


index     : integer;
index1    : integer;
temp      : string;
inp_ch    : string(1);
```

```
BEGIN
     temp := "                                        ";
     index := 1;
     index1 := 1;
     while ((index <= 40 )
             and (index <= length(ch_in.input_one))) LOOP

       if ((ch_in.input_one(index) in 'a'..'z' ) or
        (ch_in.input_one(index) in '0'..'9' )) then
           temp(index1) := ch_in.input_one(index);
           index1 := index1 + 1;
       else
          if ((character'pos(ch_in.input_one(index)) = 32)
                                 and
            ( index /= 1 ) and
            (character'pos(ch_in.input_one(index-1)) /= 32))
                                 then
              temp(index1) := ch_in.input_one(index);
              index1 := index1 + 1;
          else
              if
               ((character'pos(ch_in.input_one(index)) = 94)
                                  or
              (character'pos(ch_in.input_one(index)) = 126))
                    index1 := index1 - 1;
                    temp(index1) := ' ';
              end if;
          end if;
       end if;
       index := index + 1;
     end LOOP;
--
--   load command line
--
     ch_out.command := "         ";
     ch_out.file_name1 := "         ";
     ch_out.file_name2 := "         ";
     ch_out.command_class := ch_in.input_class;
     index := 1;
     index1 := 1;

--  loop to fill command

    while ((character'pos(temp(index)) /= 32) and
            (index1 < 9 )) LOOP

       ch_out.command(index1) := temp(index);
       index1 := index1 + 1;
       index  := index  + 1;
```

```
    end LOOP;

--  loop to fill filename 1

    index := index + 1;
    index1 := 1;

    while ((character'pos(temp(index)) /= 32)    and
           (index1 < 9 )) LOOP

       ch_out.file_name1(index1) := temp(index);
       index1 := index1 + 1;
       index  := index + 1;
    end LOOP;

--  loop to fill filename 2

    index := index + 1;
    index1 := 1;

    while ((character'pos(temp(index)) /= 32)   and
           (index1 < 9 )) LOOP

       ch_out.file_name2(index1) := temp(index);
       index1 := index1 + 1;
       index  := index  + 1;
    end LOOP;


END convert;
END files;
```

```
WITH agate, agatej, ar1, alit, alitj, strlit, util;
PACKAGE  files IS
USE agate, agatej, ar1, alit, alitj, strlit, util;


MAX_USERS              : CONSTANT integer := 3;
MAX_PROC               : CONSTANT integer := 4;
MAX_LINES              : CONSTANT integer := 100;
                                      -- max. records for file
MAX_NUMBER_OF_FILES    : CONSTANT integer := 21;
MAX_INPUT_CHSAR        : CONSTANT integer := 50;
INITIAL_FREE_SEGMENT   : CONSTANT integer := 31;
INITIAL_FREE_ENTRY     : CONSTANT integer := 0;
INITIAL_FREE_MENTOR    : CONSTANT integer := 25;
LAST_FREE_SEGMENT      : CONSTANT integer := 51;
LAST_FREE_ENTRY        : CONSTANT integer := 11;
LAST_FREE_MENTOR       : CONSTANT integer := 30;
SEGMENT_LENGTH         : CONSTANT integer := 5008;
MAX_LEVELS             : CONSTANT integer := 4;
                                     -- max. security levels
TOP_SECRET             : CONSTANT integer := 1;
SECRET                 : CONSTANT integer := 2;
CONFIDENTIAL           : CONSTANT integer := 3;
UNCLASSIFIED           : CONSTANT integer := 4;

SUBTYPE segment_number IS integer RANGE 31..51;

SUBTYPE entry_number IS integer RANGE 0..11;

SUBTYPE mentor_number IS integer RANGE 25..30;


TYPE r1_parameters IS RECORD
   entry_stack       : integer;
   mentor_stack      : integer;
   seg_number_stack : integer;
   entry_code        : integer;
   mentor_code       : integer;
   seg_number_code  : integer;
   entry_data        : integer;
   mentor_data       : integer;
   seg_number_data  : integer;
   evn_count         : integer;
   evn_count_data   : integer;
   synchr_chld_mentor : integer;
   synchr_chld_stack  : integer;
   synchr_chld_data   : integer;
END RECORD;

TYPE r1_param IS ARRAY (1..MAX_PROC ) of r1_parameters;
```

```
TYPE data_record     IS RECORD
     data1      : string(50);
END RECORD;

TYPE data_file IS ARRAY (1..MAX_LINES) of data_record;

TYPE seg_info        IS RECORD
     number          : segment_number;
     entrys          : entry_number;
     mentor          : mentor_number;
     file_name       : string(8);
     access_cla       : access_class;
     next_avail_seg : segment_number;
     next_avail_ent : entry_number;
     next_avail_men : mentor_number;
END RECORD;

TYPE segment_header IS RECORD
     max_files_stored : integer;
     next_avail_seg   : segment_number;
     next_avail_ent   : entry_number;
     next_avail_men   : mentor_number;
     max_open_seg     : segment_number;
     max_open_ent     : entry_number;
     max_open_men     : mentor_number;
END RECORD;

TYPE input_message  IS RECORD
     input_one        : string(50);
     input_result     : string(50);
     input_class      : access_class;
END RECORD;

TYPE comand_line   IS RECORD
     command          : string(8);
     file_name1       : string(8);
     file_name2       : string(8);
     command_class    : access_class;
     result           : string(50);
END RECORD;

TYPE segment_data   IS RECORD
     segm_info            : data_file;
     segm_class           : access_class;
END RECORD;

TYPE files_data IS ARRAY(0..MAX_NUMBER_OF_FILES) of
                                           seg_info;

TYPE level_record IS RECORD
```

```
    min                    : access_class;
    max                    : access_class;
END RECORD;

TYPE user_level IS ARRAY (0..MAX_LEVELS) of level_record;

TYPE user_synchro  IS RECORD
    active                 : boolean;
    seg_data               : integer;
    seg_stack              : integer;
    evc_data               : integer;
    evc_stack              : integer;
END RECORD;

TYPE users_active  IS ARRAY (1..MAX_USERS) of user_synchro;

TYPE child_resource IS RECORD

    memory                 : integer;
    processes              : integer;
    segments               : integer;
END RECORD;

PROCEDURE b24_frm_integer( in_val : in integer;
                           b24_val : out b24_type );


PROCEDURE put_ln ( ldev : in integer;
                   w_class : in access_class;
                   str : in string );


PROCEDURE blk_scr ( ldev : in integer;
                    w_class : in access_class;
                    str : in string );


PROCEDURE get_str ( ldev : in integer;
                    r_class : out access_class;
                    str : out string );


PROCEDURE put_str ( ldev : in integer;
                    w_class : in access_class;
                    str : in string );

PROCEDURE put_dec( ldev : in integer;
                   w_class : in access_class;
                   dval : in integer );

PROCEDURE put_succ( in_str : in string;
```

```
                    dec_val : in integer;
                    w_class : in access_class );

FUNCTION get_input( eco  : in boolean;
                rd_class : in access_class ) RETURN string;

PROCEDURE attach_tew( IO_PORT : in integer;
                      LDEV : in integer);

PROCEDURE attach_ter( IO_PORT : in integer;
                      LDEV : in integer) ;

PROCEDURE load_param( init : in r1_process_def;
                      ch_param : out r1_param);

PROCEDURE load_access_class( init : in r1_process_def;
                      usr_access : out user_level );


PROCEDURE load_param1(init : in r1_process_def;
                      ch_param : out r1_parameters );

PROCEDURE load_child_active(activ_usr : out users_active);

PROCEDURE lock_for_level(username : in string;
                         password : in string;
                         ch_access : in user_level;
                         ch_class : out access_class);

PROCEDURE initialize_tables(seg_table : out files_data;
                            seg_head  : out segment_header);

FUNCTION check_if_exists_file_name(seg_table : in files_data;
                      file_name : in string) RETURN boolean;

PROCEDURE convert(ch_in : in input_message;
                w_class : in access_class;
                ch_out : out comand_line );

END files;
```

APPENDIX G - SYSGEN SUBMIT FILE (SSB)

This appendix contains the description of the Sysgen Submit File used to sysgen the entire system, the commands used are :

```
bs:ld3.cmd
ks:k0.cmd
ks:k1.cmd
ks:k2.cmd
cs:v1loader.cmd;2;
ds:vilogin.cmd;2,10;
ds:nv.ds;2,5;
ds:nv.ds;5;
ds:prmain.cmd;5,0;t;
ds:puser1.cmd;5,7;
ds:prchl1.cmd;5,7,4;
ds:puser2.cmd;5,8;
ds:prchl2.cmd;5,8,4;
ds:puser3.cmd;5,9;
ds:prchl3.cmd;5,9,4;
ds:prbdos.cmd;5,10;
ds:r1trap.cmd;6;
end
```

LIST OF REFERENCES

1. Schiller, W.L., *Design and Abstract Specification of a Multics Security Kernel*, Mitre ESD-TR-77-259, Mitre Corp., Bedford, Massachusetts, November 1977.

2. Department of Defense Computer Security Center, Ft. Meade, Maryland, Report CSC-STD-001-83, *DoD Trusted Computer System Evaluation Criteria*, August 15, 1985.

3. Gemini Computers Inc., Carmel, California, *System Overview Gemini Trusted Multiple Microcomputer Base*, September 1985.

4. Boebert, E., Kain, R., and Young, B., "Trojan Horse Rolls Up to DP System," *Computerworld*, December 2, 1985.

5. Rushby and Randel, "Distributed Secure System," *Compute*, July 1985.

6. Ames, S., Gasser, M., and Schell, R., "Security Kernel Design and Implementation : An Introduction," *Computer*, July 1983.

7. Corbett, P.J., *Multilevel Secure Front End For Data Communications*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1983.

8. Cavalcanti, C.A., *Modelling of a Multilevel Secure Tactical Combat Computer System*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1986.

9. Gemini Computers Inc., Carmel, California, *Gemsos Ring 0 Users Manual for the Janus/Ada Language*, (Version 1.4), May 1986.

10. Reed, D.P. and Kanodia, R.K., "Synchronization with Eventcounts and Sequencers," *Communications of the ACM*, Vol. 22, No. 2, February 1979.

11. Gemini Computers Inc., Carmel, California, *Gemsos Ring 0 Sysgen User's Manual*, September 1985.

INITIAL DISTRIBUTION LIST

		No. Copies
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.	Chairman (Code 52) Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
4.	Computer Technology Programs (Code 37) Naval Postgraduate School Monterey, California 93943	1
5.	Gary S. Baker (Code 52Bj) Department of Computer Science Naval Postgraduate School Monterey, California 93943	5
6.	Uno R. Kodres (Code 52Kr) Department of Computer Science Naval Postgraduate School Monterey, California 93943	5
7.	Peruvian Air Force-Comando de Instruccion Ministerio de Aeronautica Lima, Peru	1
8.	Peruvian Air Force-Centro de Informatica Ministerio de Aeronautica Lima, Peru	1
9.	Mayor FAP Miguel A. Reyes Ministerio de Aeronautica-Centro de Informatica Lima, Peru	2

END
2-87
DTIC