1246

Bell Laboratories Cover Sheet for Technical Memorandum

The information contained herein is for the use of employees of Bell Laboratories and is not for publication. (See GEI 13.9-3)

Title- **The MERT Operating System**

Date- **March 22,1978**

TM- **78-3114-3**
78-1352-3

Other Keywords- UNIX
Real-Time
Multi-Environment
Interprocess Communication

Author	Location	Extension
H. Lycklama	**HO 1G-317**	**3212**
D. L. Bayer	**MH 7C-207**	**3080**

Charging Case- 39394
Filing Case- 39394-11

ABSTRACT

The MERT operating system[+] supports multiple operating system environments. Messages provide the major means of inter-process communication. Shared memory is used where tighter coupling between processes was desired. The file system was designed with real-time response being a major concern. The system has been implemented on the DEC PDP-11/45 and PDP-11/70 computers and supports the UNIX time-sharing system as well as some real-time processes.

The system is structured in four layers. The lowest layer, the *kernel*, provides basic services such as inter-process communication, process dispatching, and trap and interrupt handling. The second layer comprises privileged processes, such as I/O device handlers, the file manager, memory manager, and system scheduler. At the third layer, the supervisor processes provide the programming environments for application programs of the fourth layer. To provide an environment favorable to applications with real time response requirements, processes are permitted to control scheduling parameters such as scheduling priority and memory residency. A rich set of inter-process communication mechanisms including messages, events (software interrupts), shared memory, inter-process traps, process ports, and files, allow applications to be implemented as several independent, cooperating processes.

Some uses of the MERT operating system are discussed. A retrospective view of the MERT system is also offered. This includes a critical evaluation of some of the design decisions and a discussion of design improvements which could have been made to improve overall efficiency.

[+] A modified version of this memorandum was submitted for publication in the special issue of the BSTJ on Software. July-August, 1978.
[*] UNIX is a Trademark of Bell Laboratories.

Pages Text	23	Other	5	Total	28
No. Figures	4	No. Tables	0	No. Refs.	13

SEE REVERSE SIDE FOR DISTRIBUTION LIST

BELL TELEPHONE LABORATORIES, INC.

TM-78-3114-3
TM-78-1352-3

DISTRIBUTION
(REFER GEI 13.9-3)

COMPLETE MEMORANDUM TO

CORRESPONDENCE FILES

OFFICIAL FILE COPY
PLUS ONE COPY FOR
EACH ADDITIONAL FILING
CASE REFERENCED

DATE FILE COPY
(FORM E-1328)

10 REFERENCE COPIES

ACKROFF,JOHN M
+ARDIS,R S
BALDWIN,GEORGE L
<BARON,ROBERT V
BAUER,H C
BAYER,D L
BEAUMONT,LELAND R
BEDNAR,JOSEPH A,JR
BELL,A E
BENOWITZ,P
BERGLAND,G D
BLOSSER,PATRICK A
BOYD,GARY D
BOYLE,W S
BRADLEY,R H
BROWN,W STANLEY
CAREY,F E,JR
<CHRISTENSEN,C
CICON,J P
CLOGSTON,ALBERT M
COHEN,DAVID
CUTLER,C CHAPIN
DE ANGELIS,F E
DICKMAN,BERNARD N
DOLAN,MARIE T
DOLL,RICHARD EMIL
DOLOTTA,T A
DRAGER,JOHN
DWORAK,F S
FITCH,S M
FRASER,A G
FREEMAN,K G
FREENY,STAN L
GARST,BLAINE,JR
GILLETTE,DEAN
GLASSER,ALAN L
HAIGHT,R C
HAISCH,H F,JR
HALE,JAMES F
HALL,ANDREW D,JR
+HANNAY,N B
HAYDEN,DONALD F,JR
HEIDER,BRUCE H
HERGENHAN,C B
HESSEL,CARL M
ILLIUM,H C,JR
JUDICE,C N
KAISER,J F
KANODIA,RAJENDRA K

COMPLETE MEMORANDUM TO

KAN,HSIN-KUO
KEEFAUVER,W L
<KEESE,W M
KNEUER,JOSEPH G
KOBRAN,ALAN S
KOGELNIK,H
KRIPALANI,ANIL T
LAWLESS,WILLIAM J
LIMB,JOHN O
LIU,HUEI-CHI RICHARD
LO,DANIEL S
LUCKY,R W
LUDERER,GOTTFRIED W R
<LYCKLAMA A NYEHOLT,H
MANCUSI,M D
MARANZANO,J F
<MC DONALD,H S
MCCULLOCH,JOHN W
MCILROY,M DOUGLAS
<MENNINGER,R E
METZLER,HELEN M
MILLER,STEWART E
MOLLENAUER,JAMES F
MORGAN,SAMUEL P
MOSTER,C R
NINKE,WILLIAM H
OLSON,ELIZABETH
OTT,PETER A
PAPPAS,A L
PASTERNACK,G
PENZIAS,A A
+PRIM,R C
RABIN,JONAS
REDMOND,K L
RENDER,DONALD J
RICE,DONALD W
<ROBERTS,CHARLES S
ROCHKIND,M M
RODRIGUEZ,ERNESTO J
ROSENFELD,PETER E
SALTZBERG,B R
SANTILLO,JAMIE ANNETTE
SCHAEFER,J W
SCHLANGER,G GARY
SCHWEBEL,JOHN C
SHANNON,MARTIN J,JR
SHIRTZ,ANN L
SLICHTER,W P
SPIVACK,JERRY
STURMAN,JOEL N
TAGUE,BERKLEY A
TERRY,M E
<TEWKSBURY,S F
<THOMPSON,J S
THOMPSON,RICHARD A
THORNTON,PETER P
TING,DENNIS WAY
TRIOLO,VICTOR J
TUKEY,JOHN W
TUTELMAN,DAVID M
UNGAR,DAVID M

COMPLETE MEMORANDUM TO

USAS,ALAN M
WAGNER,R L
WATSON,D S
WELLER,D R
WILD,J CHRISTIAN,JR
WOLONTIS,V MICHAEL
<YAMIN,ELAINE E
ZELASKO,JO ANN F
ZISLIS,PAUL M
119 NAMES

COVER SHEET ONLY TO

CORRESPONDENCE FILES

4 COPIES PLUS ONE
COPY FOR EACH FILING
CASE

AAGESEN,JOHN
ABATEMARCO,TERESA M
ABATE,JOSEPH
ABBRECHT,RICHARD L
ACKERMAN,A FRANK
AHO,ALFRED V
AHRENS,RAINER B
AHUJA,SUDHIR R
ALBERTS,BARBARA A
ALCALAY,D
ALEXIS,A D,JR
<ALLES,H G
ALLISON,C E,JR
AMITAY,N
AMOSS,JOHN J
ANDERSON,C R
ANDERSON,JOSEPH N,JR
ANDERSON,MILTON M
ANDERSON,S E
ANDERSON,ROBERT V
ANDERSON,W A
ANTOLICK,DAVID R
ANTONELLI,PHYLLIS
ARCHER,VANCE D,III
ARNDT,DENNIS L
ARNOLD,GEORGE W
ARNOLD,S L
ARNOLD,THOMAS F
ARSENAULT,J E
ARTHURS,E
ARTIS,H P
ASTHANA,ABHAYA
ATKINS,JOAN
AXELSON,A L
BABECKI,GLENN R
BABU,RAJESH RATILAL
BACCASH,JEANNE M
BACH,MAURICE J
BACKUS,C F,SR
BAGLEY,JOHN D

COVER SHEET ONLY TO

BAIN,WILLIAM LAMAR,JR
<BALDWIN,GARY L
BALENSON,CHRISTINE M
BALL,MARSHALL
BANKER,KIM K
BARCLAY,DAVID K
BARR,W J
BASEIL,RICHARD J
BATISTONI,F J
BATTAGLIA,FRANCES
BAUER,ANDREW E
BAUER,BARBARA T
BAUER,HELEN A
<BAUGH,C R
BAUMAN,STEVEN M
BAXTER,LESLIE A
BECKER,JACOB I
BECKER,RICHARD A
BECKETT,J T
BECRAFT,A D
BEL BRUNO,KATHLEEN A
<BENES,V E
BENISCH,JEAN
BENJAMIN,O J
BENNETT,RICHARD L
BENNETT,WILLIAM C
<BENNING,R D
BERGERON,R F,JR
BERNSTEIN,DANIELLE R
BERNSTEIN,L
BERRYMAN,R D
<BEYER,JEAN-DAVID
BEYLER,ERIC
BHATIA,RAJIV
BIANCHI,M H
BIAZZO,M R
BICKFORD,NEIL B
BILOWOS,R M
BIRCHALL,R H
BIREN,IRMA B
BISHOP,J DANIEL
BITTNER,BARBARA B
BITTRICH,MARY E
BLANCHARD,JOETTE D
BLAZIER,S D
BLEICHER,E
BLINDER,S
BLINN,J C
BLISS,ROBERT H
BLUE,JAMES L
BLUMER,THOMAS P
BLUM,MARION
BOCKUS,ROBERT J
BOCK,NANCY E
BODDIE,JAMES R
BODEN,F J
BOGART,F J
BOIANO,ROSEMARY
BOIVIE,RICHARD H
BONACHEA,R N
BONANNI,L E

COVER SHEET ONLY TO

BOND,HOLTON C,JR
BOPP,J ROBERT
BORG,KEVIN E
BOSE,DEBASISH
BOSTON,RONALD E
BOSWORTH,R H
BOTHUR,R H
BOURNE,STEPHEN R
BOWEN,E G
BOWYER,L RAY
BOYCE,C D
BOYCE,W M
BOYD,K W
BOYLE,GERALD C
BRADFORD,EDWARD G
BRADLEY,M HELEN
BRAINARD,R C
BRANDAUER,C M
BRANDT,RICHARD B
BRANTLEY,JOAN T
BREECE,H T,3RD
BRESLER,RENEE A
BRITT,WARREN D
BRONSTEIN,N
BROSS,JEFFREY D
BROWNE,PAUL N
BROWN,C W
BROWN,E L
BROWN,EARL F
BROWN,ROBIN L
BROWN,W R
BUCK,I D
BULFER,A F
BURCH,GLORIA J
BURGESS,JOHN T,JR
BURG,F M
BURKE,R J
BURNETTE,W A
BURNETT,DAVID S
BURNET,ROSE M
BURNS,GARY J
BUROFF,STEVEN J
BURROWS,THOMAS A
BURR,DAVID JOSEPH
BUTLER,D F
<BUTZIEN,PAUL E
BYORICK,ROBERT S
BYRNE,EDWARD R
CALLAHAN,R L
CAMLET,J V,JR
CAMPBELL,JERRY H
CAMPBELL,MICHAEL P
CANADAY,RUDD H
CANDELIERI,ANTHONY V
CANDREA,RONALD D
CANDY,J C
CANNON,LAYNE W
CARBONE,J T
CARDOZA,WAYNE M
CAREY,J E
CARRAN,JOHN H
...
1175 TOTAL

+ NAMED BY AUTHOR   > CITED AS REFERENCE   < REQUESTED BY READER   (NAMES WITHOUT PREFIX
WERE SELECTED USING THE AUTHOR'S SUBJECT OR ORGANIZATIONAL SPECIFICATION AS GIVEN BELOW)

MERCURY SPECIFICATION..

COMPLETE MEMO TO:

135-CPB   13-DIR   11-EXD   15-EXD   16-EXD   135-SUP   127-SUP   311-SUP   10-EXD   12-EXD
13-EXD   17-EXD   31-EXD   823-SUP   312-SUP   32-EXD   1352   3114   3111   3124

COVER SHEET TO:

135   127   8234   9152   312

COTCOS = THEORY OF COMPUTER OPERATING SYSTEMS
UNOS* = UNIX OPERATING SYSTEM: GENERAL OR SURVEY DOCUMENTS

HO CORRESPONDENCE FILES
HO 5C101

TM-78-3114-3
TOTAL PAGES 28

TO GET A COMPLETE COPY:

1. BE SURE YOUR CORRECT ADDRESS IS GIVEN ON THE OTHER SIDE.
2. FOLD THIS SHEET IN HALF WITH THIS SIDE OUT AND STAPLE.
3. CIRCLE THE ADDRESS AT RIGHT. USE NO ENVELOPE.

PLEASE SEND A COMPLETE

( ) MICROFICHE COPY   ( ) PAPER COPY

TO THE ADDRESS SHOWN ON THE OTHER SIDE.

Bell Laboratories

Subject: The MERT Operating System [+]
Case- 39394 -- File- 39394-11

date: March 22,1978

from: H. Lycklama
D. L. Bayer

TM: 78-3114-3
78-1352-3

MEMORANDUM FOR FILE

1. Introduction

As operating systems become more sophisticated and complex, providing more and more services for the user, they become increasingly difficult to modify and maintain. Fixing a "bug" in some part of the system may very likely introduce another "bug" in a seemingly unrelated section of code. Changing a data structure is likely to have major impact on the total system. It has thus become increasingly apparent over the past years that adhering to the principles of structured modularity (1),(2) is the correct approach to building an operating system. The objective of the MERT system has been to extend the concept of a process into the operating system, factoring the traditional operating system functions into a small kernel surrounded by a set of independent cooperating processes. Communication between these processes is accomplished primarily through messages. Messages define the interface between processes and reduce the number of ways a bug can be propagated through the system.

The MERT kernel establishes an extended instruction set via system primitives vis-a-vis the virtual machine approach of CP 67. Operating systems are implemented on top of the MERT kernel and define the services available to user programs. Communication and synchronization primitives and shared memory permit varying degrees of co-operation between independent operating systems. An operating system functionally equivalent to the UNIX[*] time sharing system has been implemented to exercise the MERT executive and provide tools for developing and maintaining other operating system environments. An immediate benefit of this approach is that operating system environments tailored to the needs of specific classes of real-time projects can be implemented without interfering with other systems having different objectives.

One of the basic design goals of the system was to build modular and independent processes having data structures and tables which are known only to the particular process. Fixing a "bug" or making major internal changes in one process does not affect the other processes with which it communicates. The work described here builds on previous operating system designs described by Dijkstra (1) and Brinch Hansen (2). The primary differences between this system and previous work lies in the rich set of inter-process communication techniques and the extension of the concept of independent modular processes, protected from other processes in the system, to the basic I/O and real-time processes. It can be shown that messages are not an adequate communication path for some real-time problems (3). Controlled access to shared memory, and software-generated interrupts are often required to maintain the integrity of a real-time system. The communication primitives were selected in an attempt to balance the need for protection with the need for real-time response. The primitives include event flags, message buffers, inter-process system traps, process ports and shared segments.

[+] A modified version of this memorandum was submitted for publication in the special issue of the BSTJ on Software, July, August, 1978.

[*] UNIX is a Trademark of Bell Laboratories.

One of the major influences on the design of the MERT system came from the requirements of various application systems at Bell Laboratories. They made use of imbedded minicomputers to provide support for development of application programs and for controlling their specific application. Many of these projects had requirements for real-time response to various external events. Real-time can be classified into two categories. One flavor of real-time requires the collection of large amounts of data. This necessitates the implementation of large and contiguous files and asynchronous I/O. The second flavor of real-time demands quick response to hardware-generated interrupts. This necessitates the implementation of processes locked in memory. Yet another requirement for some applications was the need to define a more-controlled environment with better control over a program's virtual address space lay-out than that provided in a general-purpose time-sharing environment.

This paper gives a detailed description of the system design including the kernel, and a definition and description of processes and of segments. A detailed discussion of the communication primitives follows. The structure of the file system is then discussed along with how the file manager and time-sharing processes make use of the communication primitives.

A major portion of this paper deals with a critical retrospective on the MERT system. This includes a discussion of features of the MERT system which have been used by various projects within the Bell System. Some trade-offs are given that have been made for efficiency reasons thereby sacrificing some protection. Some operational statistics are also included here. The pros and cons of certain features of the MERT operating system are discussed in detail. The portability of the operating system as well as user software is currently a topic of great interest. The prospects of the portability of the MERT system are described. Finally, we discuss some features of the MERT system which could have been implemented differently for the sake of efficiency.

2. Hardware Requirements

The MERT system currently runs on the DEC PDP-11/45 and PDP-11/70 computers (4). These computers provide an eight-level static priority interrupt structure with priority levels numbered from 0 (lowest) to 7 (highest). Associated with the interrupt structure is the programmed interrupt register which permits the processor to generate interrupts at priorities of one through seven. The programmed interrupt serves as the basic mechanism for driving the system.

The PDP-11 computer is a 16-bit word machine with a direct address space of 32K words. The memory management unit on the PDP-11/45 and PDP-11/70 computers provides a separate set of address mapping and access control registers for each of the processor modes: kernel, supervisor and user. Furthermore, each virtual address space can provide separate maps for instruction references (called I-space) and data references (D-space). The MERT system makes use of all three processor modes (kernel, supervisor and user) and both the instruction and data address spaces provided by these machines.

3. System Design

Processes are arranged in four levels of protection (see Figure 1). The lowest level of the operating system structure, called the kernel, allocates the basic computer resources. These resources consist of memory, segments, the CPU, and interrupts. All process dispatching, including interrupt processing, is handled by the kernel dispatcher. The kernel is the most highly privileged system component and therefore must be the most reliable.

The second level of software consists of kernel-mode processes which comprise the various I/O device drivers. Each process at this level has access to a limited number of I-space base registers in the kernel mode, providing a firewall between it and sensitive system data accessible only using D-space mode. Within this level processes are linked onto one of five priority lists. These lists correspond to the processor priority required while the process is executing. Three kernel processes must exist for the system to function:

1. The file manager is required since all processes are derived from files.
2. The swap process is required to move segments between secondary storage and main memory.
3. The root process is required to carry out data transfers between the file manager and the disk.

Since the same device usually contains both the swap area and root file system, one process usually serves for both 2. and 3.

At the third software level are the various operating system supervisors which run in supervisor mode. These processes provide the environments which the user sees and the interface to* the basic kernel services. All processes at this level execute at a processor priority of either one or zero. A software priority is maintained for the supervisor by the scheduler process. Two supervisor processes are always present: the process manager which creates all new processes and produces post mortem dumps of processes which terminate abnormally, and the time sharing supervisor.

At the fourth level are the various user procedures which execute in user mode under control of the supervisory environments. The primitives available to the user are provided by the supervisory environments which process the user system calls. Actually the user procedure is merely an extension of the supervisor process. This is the highest level of protection provided by the computer hardware.

4. Definitions

4.1. Segments

A logical segment is a piece of contiguous memory, 32 to 32K 16-bit words long, which can grow in increments of 32 words. Associated with each segment are an internal segment identifier (ID) and an optional global name. The segment identifier is allocated to the segment when it is created and is used for all references to the segment. The global name uniquely defines the initial contents of the segment. A segment is created on demand and disappears when all processes which are linked to it are removed. The contents of a segment may be initialized by copying all or part of a file into the segment. Access to the segment can be controlled by the creator (parent) as follows:

1) The segment can be private - that is, available only to the creator.
2) The segment can be shared by the creator and some or all of its descendents (children). This is accomplished by passing the segment ID to a child.
3) The segment can be given a name which is available to all processes in the system. The name is a unique 32-bit number which corresponds to the actual location on secondary storage of the initial segment data. Processes without a parent-child relationship can request the name from the file system and then attempt to create a segment with that name. If the segment exists, the segment ID is returned and the segment user count is incremented. Otherwise the segment is created and the process initializes it.

4.2. Processes

A process consists of a collection of related logical segments executed by the processor. Processes are divided into two classes, kernel processes and supervisor processes, according to the mode of the processor while executing the segments of the process.

* The time sharing supervisor can create a new process consisting of an exact copy of itself.

4.2.1. Kernel Processes

Kernel processes are driven by software and hardware interrupts, execute at processor hardware priority 2 to 7, are locked in memory, and are capable of executing all privileged instructions. Kernel processes are used to control peripheral devices and handle functions with stringent real-time response requirements.

The virtual address space of each kernel process begins with a short header which defines the virtual address space and various entry points (see Figure 2). Up to 12K words (base registers 3 - 5) of instruction space and 12K words of data space are available. All kernel processes share a common stack and can read and write the I/O device registers.

To reduce duplication of common subprograms used by independent kernel processes and to provide common data areas between independent cooperating kernel and supervisor processes, three mechanisms for sharing segments are available.

The first type of shared segment, called the system library, is available to all kernel processes. The routines included in this library are determined by the system administrator at system generation time. The system library begins at virtual address 140000(8) (base register 6) and is present whether or not it is used by any kernel processes.

The second type of shared segment, called a public library, is assigned to base registers four or five of the process instruction space. References to routines in the library are satisfied when the process is formed, but the body of the segment is loaded into memory only when the first process which accesses it is loaded.

A third sharing mechanism allows a parent to pass the ID of a segment that is included in the address space of a kernel process when it is created. This form of sharing is useful when a hierarchy of cooperating processes is invoked to accomplish a task.

4.2.2. Supervisor Processes

All processes which execute in supervisor mode and user mode are called supervisor processes. These processes run at processor priority zero or one and are scheduled by the kernel scheduler process. The segments of a supervisor may be kept in memory, providing response on the order of several milliseconds, or supervisor segments may be swappable, providing a response time of hundreds of milliseconds.

The virtual address space of a supervisor process consists of 32K words of instruction space and 32K words of data space in both supervisor and user modes (see Figure 3). Of these 128K, at least part of each of three base registers (a total of 12K) must be used for access to:

1) the process control block (PCB), a segment typically 160 words long, which describes the entire virtual address space of the process to the kernel and provides space to save the state of the process during a context switch. The PCB also includes a list of capabilities which define the range of abilities of the process.
2) the process supervisor stack and data segment.
3) the read-only code segment of the supervisor.

The rest of the virtual address space is controlled by the supervisor. The primitives available to supervisor processes include the ability to control the virtual address space (both supervisor and user) which can be accessed by the process.

4.3. Capabilities

Associated with each supervisor process is a list of keys, each of which allows access to one object. The capability key must be passed as an argument in all service requests on objects. Each key is a concatenation of the process ID of the creator of the object and a bit pattern, defined by the creator, which describes allowed operations on the object. The capability list (C-list) for each supervisor process resides in the PCB and is maintained by the kernel through *add* and *delete* capability messages to the memory manager. A special variation of the send message primitive copies the capability from the PCB into the body of a message, preventing corruption of the capability mechanism.

Capabilities are used by the file manager to control access to files. The capability for a file is granted upon opening the file. A read or write request is validated by decoding the capability into a fourteen bit object descriptor (file descriptor) and a two bit permission field. The capability is removed from the process C-list when the file is closed.

5. The Kernel

The concept of an operating system nucleus or kernel has been used in several systems. Each system has included a different set of logical functions (5), (6). The MERT kernel is to be distinguished from a *security* kernel. A *security* kernel provides the basis of a secure operating system environment.

The basic kernel provides a set of services available to all processes, kernel and supervisor, and maintains the system process tables and segment tables. Included as part of the kernel are two special system processes, the memory manager and the scheduler. These are distinguished from other kernel processes in that they are bound into the basic kernel address space and do not require the set up of a base register when dispatched to.

5.1. Kernel Modules

The kernel consists of a process dispatcher, a trap handler, and routines (procedures) which implement the system primitives. Approximately 4K words of code are dedicated to these modules.

The *process dispatcher* is responsible for saving the current state and setting up and dispatching to all kernel processes. It can be invoked by an interrupt from the programmed interrupt register, an interrupt from an external device, or an inter-process system trap from a supervisor process (an EMT trap).

The *trap handler* fields all traps and faults and, in most cases, transfers control to a trap handling routine in the process which caused the trap or fault.

The *kernel primitives* can be grouped into eight logical categories. These categories can be subdivided into those which are available to all processes and others which are available only to supervisor processes. The primitives which are available to all processes are:

1) Interprocess communication and synchronization primitives. These include sending and receiving of messages and events, waking up processes which are sleeping on a bit pattern, and setting the sleep pattern.
2) Attaching to and detaching from interrupts.
3) Setting a timer to cause a time-out event.
4) Manipulation of segments for the purposes of input/output. This includes locking and unlocking segments and marking segments altered.
5) Setting and getting the time of day.

The primitives available only to supervisor processes are:

6) Primitives which alter the attributes of the segments of a process. These primitives include creating new segments, returning segments to the system, adding and deleting segments from the process address space, and altering the access permissions.
7) Altering scheduler-related parameters by roadblocking, changing the scheduling priority, or making the segments of the process nonswap or swappable.
8) Miscellaneous services such as reading the console switches.

5.2. Kernel System Processes

Closely associated with the kernel are the memory management and scheduler processes. These two processes are special in that they reside in the kernel address space, permanently. In all other respects they follow the discipline established for kernel processes.

5.2.1. Memory Manager

The memory manager is a special system process. It communicates with the rest of the system via messages and is capable of handling four types of requests:

1) Setting the segments of a process into the active state, making space by swapping or shifting other segments if necessary.
2) Loading and locking a segment contiguous with other locked segments to reduce memory fragmentation.
3) Deactivating the segments of a process.
4) Adding and deleting capabilities from the capability list in a supervisor process PCB.

5.2.2. Scheduler

The scheduler is the second special system process and is responsible for scheduling all supervisor processes. The main responsibility of the scheduler is to select the next process to be executed. The actual loading of the process is accomplished by the memory manager.

5.3. Dispatcher Mechanism

The system maintains seven process lists, one for each processor priority at which software interrupts can be triggered using the programmed interrupt register. All kernel processes are linked into one of the six lists for processor priorities two through seven; all supervisor processes are linked to the processor priority one list. The occurrence of a software interrupt at priorities two through seven causes the process dispatcher to search the corresponding process list and dispatch to all processes which have one or more event flags set. The entire list is searched for each software interrupt.

5.4. Scheduling Policy

All software interrupts at processor priority one, which are not for the currently active process, cause the dispatcher to send a wakeup event to the scheduler process. The scheduler uses a byte in the system process tables to maintain the scheduling priority of each process. This byte is manipulated by the scheduler as follows:

1) Incremented when a process receives an event.
2) Increased by ten when awakened by a kernel process.
3) Decremented when the process yields control due to a roadblock system call.
4) Lowered according to an exponential function each successive time the process uses its entire time slice (becomes compute bound).

The process list is searched for the highest priority process which is ready to run, and if this process has higher priority than the current process, the new process will preempt the current process.

To minimize thrashing and swapping, the scheduler uses a "will receive an event soon" flag which is set by the process when it roadblocks. This flag is typically set when a process roadblocks awaiting completion of I/O which is expected to finish in a short time relative to the length of the time slice. The scheduler will keep the process in memory for the remainder of its time slice. When memory becomes full and all processes which require loading are of sufficiently low priority, the scheduler stops making load requests until one of the processes being held runs out of its time slice.

6. Inter-Process Communication

A structured system requires a well-defined set of communication primitives to permit inter-process communication and synchronization. The MERT system makes use of the following communication primitives to achieve this end:

(1) event flags
(2) message buffers
(3) EMT traps
(4) shared memory
(5) files
(6) process ports

Each of these is discussed in further detail here.

6.1. Event Flags

Event flags are an efficient means of communication between processes for the transfer of small quantities of data. Of the 16 possible event flags per process, eight are predefined by the system for the following events: wakeup, timeout, message arrival, hangup, interrupt, quit, abort and initialization. The other eight event flags are definable by the processes using the event flags as a means of communication. Events are sent by means of the kernel primitive:

```
event(procid, event)
```

When control is passed to the process at its event entry point the event flags are in its address space.

6.2. Message Buffers

The use of message buffers for inter-process communication was introduced in the design of the RC4000 operating system (2). The SUE project (7) also used a message sending facility and the related device called a mailbox to achieve process synchronization. We introduce here a set of message buffer primitives which provide an efficient means of inter-process communication and synchronization.

A kernel pool of message buffers is provided, each of which may be up to a multiple of seven times 16 words in size. Each message consists of a six word header and the data being sent to the receiving process. The format of the message is specified in Figure 4. The primitives available to a process consist of:

```
alocmsg(nwords)
queuem(message)
queuemn(message)
dequeuem(process)
dqtype(process)
messink(message)
freemsg(message)
```

To open a communication channel between two processes P1 and P2, P1 must allocate a message buffer using *alocmsg*, fill in the appropriate data in the message header and data areas and then send the message to process P2 using *queuem*. Efficiency is achieved by allowing P1 to send multiple messages before waiting for an acknowledgement (answer). The acknowledgement to these messages is returned in the same buffer by means of the *messink* primitive. The message buffer address space is freed up automatically if the message is an acknowledgement to an acknowledgement. Buffer space may also be freed explicitly by means of the *freemsg* primitive. When no answer is expected back from a process, the *queuemn* primitive is used.

Synchronization is achieved by putting the messages on P2's message input queue using the *link* word in the message header and sending P2 a message event flag. This will immediately invoke the scheduling of process P2 if it runs at a higher priority than P1. Process P1 is responsible for filling in the *from process* number, the *to process* number, the *type* and the *identifier* fields in the message header. The *type* field specifies which routine P2 must execute to process the message. A type of '-1' is reserved for acknowledgement messages to the original sender of the message. The status of the processed message is returned in the *status* field of the message header, a non-zero value indicating an error. The status of -1 is reserved for use by the system to indicate that process P2 does not exist or was terminated abnormally while processing the message. The *sequence number* field is used solely for debugging purposes. The *identifier* field may be planted by P1 to be used to identify and verify acknowledgement messages. This word is not modified by the system.

Process P2 achieves synchronization by waiting for a message. In general a process may receive any message type from any process by means of the *dequeuem* primitive. However P2 may request a message type by means of *dqtype* in order to process messages in a certain sequence for internal process management. In each case the kernel primitive will return a success/fail condition. In the case of a fail return, P2 has the option of roadblocking to wait for a message event or of doing further processing and looking for an input message at a later time.

6.3. EMT Traps

The emulator trap (EMT) instruction is used not only to implement the system primitives, but also to provide a mechanism by which a supervisor and kernel process can pass information. The supervisor process passes the process number of the kernel process with which it would like to communicate to the kernel. The kernel then dispatches to the kernel process through its EMT entry point, passing the process number of the calling supervisor process and a pointer to an argument list. The kernel process will typically access data in the supervisor process address space by setting part of its virtual address space to overlap that of the supervisor. This method of communication is used mainly to pass characters from a time-sharing user to the kernel process which controls communications equipment.

6.4. Shared Memory

Supervisor processes may share memory by means of named as well as unnamed segments. Segments may be shared on a supervisor as well as a user level. In both cases pure code is shared as named segments. In the case of a time-sharing supervisor (described in Section 8), a segment is shared for I/O buffers and file descriptors. A shared segment is also used to implement the concept of a *pipe* (8), which is an inter-process channel used to communicate streams of data between related processes. At the user level, related processes may share a segment for the efficient communication of a large quantity of data. For related processes, a parent process may set up a shareable segment in its address space and restrict the access permissions of all child processes to provide a means of protecting shared data. Facilities are also provided for sharing segments between unrelated supervisors and between kernel and supervisor processes.

6.5. Files

The file system has a hierarchical structure equivalent to the UNIX file system (8) and as such has certain protection keys (see Section 7). Most files have general read/write permissions and the contents are shareable between processes.

In some cases, the access permissions of the file may themselves serve as a means of communication. If a file is created with read/write permissions for the owner only, another process may not access this file. This is a means of making that file name unavailable to a second process.

6.6. Process Ports

Knowing the identity of another process gives a process the ability to communicate with it. The identity of certain key processes must be known to all other processes at system startup time to enable communication. These globally known processes include the scheduler, the memory manager, the process manager, the file manager and the swap device driver process. These comprise a sufficient set of known processes to start up new processes which may then communicate with the original set.

Device driver processes are created dynamically in the system. They are in fact created, loaded and locked in memory upon opening a "device" file (see Section 7). The identity of the device driver process is returned by the process manager to the file manager which in turn may return the identity to the process which requested the opening of the "device" file. These processes are referred to as "external" processes by Brinch Hansen (2).

The above process communication primitives do not satisfy the requirements of communication between unrelated processes. For this reason the concept of process ports has been introduced. A process port is a globally known "device" (name) to which a process may attach itself in order to communicate with "unknown" processes. A process may connect itself to a port, disconnect itself from a port or obtain the identity of a process connected to a specific port. Once a process identifies itself globally by connecting itself to a port, other processes may communicate with it by sending messages to it through the port. The port thus serves as a two-way communication channel. It is a means of communication for processes which are not descendents of each other.

7. File System

The multi-environment as well as the real-time aspects of the MERT system require that the file system structure be capable of handling many different types of requests. Time-sharing applications require that files be both dynamically allocatable and dynamically growable. Real-time applications require that files be large, and possibly contiguous; dynamic allocation and growth are usually not required.

For data base management systems, files may be very large and it is often advantageous that files be stored in one contiguous area of secondary storage. Such large files are efficiently described by a file-map entry which consists of starting block number and number of consecutive blocks (a two-word extent). A further benefit of this allocation scheme is that file accesses require only one access to secondary storage. Another commonly used scheme, using indexed pointers to blocks of a file in a file-map entry, may require more than one access to secondary storage to read or write a block of a file. However, this latter organization is usually quite suitable for time-sharing applications. The disadvantage of using two-word extents in the file-map entry to describe a dynamic time-sharing file is that this may lead to secondary storage fragmentation. In practice the efficient management of the in-core free extents reduces storage fragmentation significantly.

Three kinds of files are discernible to the user: ordinary disk files, directories and special files. The directory structure is identical to the UNIX file system directory structure. Directories provide the mapping between the names of files and the files themselves and impose a hierarchical naming convention on the files. A directory entry contains only the name of the file and a file identifier which is essentially a pointer to the file-map entry (inode) for that file. A file may have more than one link to it, thus enabling the sharing of files.

Special files in the MERT system are associated with each I/O device. The opening of a special file causes the file manager to send a message to the process manager to create and load the appropriate device driver process and lock it in memory. Subsequent reads and writes to the file are translated into read/write messages to the corresponding I/O driver process by the file manager process.

In the case of ordinary files, the contents of a file are whatever the user puts in it. The file system process imposes no structure on the contents of the file.

The MERT file system distinguishes between contiguous files and other ordinary files. Contiguous files are described by one extent and the file blocks are not freed until the last link to the file is removed. Ordinary files may grow dynamically using up to 27 extents to describe their secondary storage allocation. To minimize fragmentation of the file system a growing file is allocated 40 blocks at a time. Unused blocks are freed when the file is closed.

The list of free blocks of secondary storage is kept in memory as a list of the 64 largest extents of contiguous free blocks. Blocks for files are allocated and freed from this list using an algorithm which minimizes file system fragmentation. When freeing blocks, the blocks are merged into an existing entry in the free list if possible, otherwise placed in an unused entry in the free list, or failing this, replace an entry in the free list which contains a smaller number of free blocks.

The entries which are being freed or allocated are also added to an update list in memory. These update entries are used to update a bit map which resides on secondary storage. If the in-core free list should become exhausted, the bit map is consulted to re-create the 64 largest entries of contiguous free blocks. The nature of the file system and the techniques used to reduce file system fragmentation ensure that this is a very rare occurrence.

Very active file systems consisting of many small time-sharing files may be compacted periodically by a utility program to minimize file system fragmentation still further. File system storage fragmentation actually only becomes a problem when a file is unable to grow dynamically having used up all 27 extents in its file map entry. Normal time-sharing files do not approach this condition.

Communication with the file system process is achieved entirely by means of messages. The file manager can handle 25 different types of messages. The file manager is a kernel process using both I and D space. It is structured as a task manager controlling a number of parallel co-operating tasks which operate on a common data base and which are not individually preemptible. Each task acts on behalf of one incoming message and has a private data area as well as a common data area. The parallel nature of the file manager ensures efficient handling of the file system messages. The mode of communication, message buffers, also guarantees that other processes need not know the details of the structure of the file system. Changes in the file system structure are easily implemented without affecting other process structures.

8. A Time-Sharing Supervisor

The first supervisor process developed for the MERT system was a time-sharing supervisor logically equivalent to the UNIX time-sharing system (8). The UNIX supervisor process was implemented using messages to communicate with the file system manager. This makes the UNIX supervisor completely independent of the file system structure. Changes and additions can then be made to the file system process as well as the file system structure on secondary storage without affecting the operation of the UNIX supervisor.

The structure of the system requires that there be an independent UNIX process for each user who "logs in". In fact a UNIX process is started up when a "carrier-on" transition is detected on a line which is capable of starting up a user.

For efficiency purposes the code of the UNIX supervisor is shared among all processes running in the UNIX system environment. Each supervisor has a private data segment for maintaining the process stack and hence the state of the process. For purposes of communication one large data segment is shared among all UNIX processes. This data segment contains a set of shared buffers used for system side-buffering and a set of shared file descriptors which define the files that are currently open.

The sharing of this common data segment does introduce the problem of critical regions, i.e. regions during which common resources are allocated and freed. The real-time nature of the system means that a process could be preempted even while running in a critical region. To ensure that this does not occur, it is necessary to inhibit preemption during a critical region and then permit preemption again upon exiting from the critical region. This also guarantees that the delivery of an event at a higher hardware priority will not cause a critical region to be reentered. Note that a semaphore implemented at the supervisor level cannot prevent such reentry unless events are inhibited during the setting of the semaphore.

The UNIX supervisor makes use of all of the communication primitives discussed previously. Messages are used to communicate with the file system process. Events and shared memory are used to communicate with other UNIX processes. Communication with character device driver processes is by means of EMT traps. Files are used to share information among processes. Process ports are used in the implementation of an error logger process to collect error messages from the various I/O device driver processes.

The entire code for the UNIX supervisor process consists of 8K words. This includes all of the standard UNIX system routines as well as the many extra system routines which have been added to the UNIX/MERT supervisor. The extra system routines make use of the unique features available under MERT. These include the ability to:

1) create a new environment
2) send and receive messages
3) send and receive events
4) set up shared segments
5) invoke new file system primitives such as allocate contiguous files
6) set up and communicate with process ports
7) initiate physical and asynchronous I/O.

All memory management and process scheduling functions are performed by the kernel.

9. Real Time Aspects

Several features of the MERT architecture make it a sound base on which to build real-time operating systems. The kernel provides the primitives needed to construct a system of cooperating, independent processes, each of which is designed to handle one aspect of the larger real-time problem. The processes can be arranged in levels of decreasing privilege depending on the response requirements. Kernel processes are capable of responding to interrupts within 100 microseconds, non-swap supervisor processes can respond within a few milliseconds, and swap processes can respond in hundreds of milliseconds. Shared segments can be used to pass data between the levels and to insure that the most up-to-date data is always available. This is sufficient to solve the data integrity problem discussed by Sorenson (3).

The system provides a low resolution interval timer which can be used to generate events at any multiple of 1/60th of a second up to 65535. This is used to stimulate processes which update data bases at regular intervals or time I/O devices. Since the timer event is an interrupt, supervisor processes can use it to subdivide a time slice to do internal scheduling.

The preemptive priority scheduler and the control over which processes are swappable allow the system designer to specify the order in which tasks are processed. Since the file manager is an independent process driven by messages, all processes can communicate directly with it, providing a limited amount of device independence. The ability to store a file on a contiguous area of secondary storage is aimed at minimizing access time. Finally, the availability of a sophisticated time-sharing system in the same machine as the real-time operating system provides powerful tools which can be exploited in designing the man-machine interface to the real-time processes.

10. Process Debugging

One of the most useful features of the system is the ability to carry on system development while users are logged in. New I/O drivers have been debugged and experiments with new versions of the time-sharing supervisor have been performed without adversely affecting the user community.

Three aspects of the system make this possible:

1) Processes can be loaded dynamically.
2) Snap shot dumps of the process can be made using the time-sharing supervisor.
3) Processes are gracefully removed from the system and a core dump produced on the occurrence of a "break point trap".

As an example, we recently interfaced a PDP-11/20 to our system using an interprocessor DMA link. During the debugging of the software, the two machines would often get out of phase leading to a break-down in the communication channel. When this occurred, a dump of the process handling the PDP-11/45 end of the link was produced, a core image of the PDP-11/20 was transmitted to the PDP-11/45, and the two images were analyzed using a symbolic debugger running under the time-sharing supervisor. When the problem was fixed a new version of the kernel-mode link process was created, loaded, and tested. Turn around time in this mode of operation is measured in seconds or minutes.

11. MERT-Based Projects

A number of PDP-11 based minicomputer systems have taken advantage of the MERT system features to meet their system specifications. In this section a survey of these projects, together with the features used is presented.

11.1. TSPS Evaluation and Testing

TSPS evaluation and testing is being done by using a computer for voice recording and voice playback. The collection or output of data is at the rate of 2000 16-bit words per second. Data recording sessions typically last for several minutes. The solution chosen was to use a pre-allocated contiguous file for data collection. A kernel process was used to provide the real-time response necessary. The name of the file used for data collection and various other parameters are passed to the kernel process from the user process which created (spawned) the kernel process. The kernel process sends an "open file" message to the file manager process and then starts the collection of data. To ensure uninterrupted data collection, the data is double-buffered, i.e. when one 1024 word buffer is filled, a message is sent to the file manager to write out the buffer; meanwhile the kernel process continues to collect data in the second buffer under interrupt control. The "write file" message to the file manager process is handled in parallel. When the write is complete, an acknowledgement message is returned to the kernel process. Upon completion of data collection by the kernel process, it sends a message to its parent user process with information concerning the status of the data transfer. The user is now back in a "UNIX-like" time-sharing environment and can do data analysis using the tools available under the UNIX system.

11.2. AEMIS

The AEMIS project is a Management Information System for an Automatic Call Distribution Center for ESS. A special kernel process controls a DR11C device to communicate with up to six #1 ESS machines. As this word-at-a-time device does not produce interrupts, it is necessary to periodically scan the status of each data link and read each channel with input data. The kernel driver process invokes a time-out event every 33 milli-seconds and wakes up the higher-level processes which are waiting to read this device. At the user program level, events are used for communication of small amounts of data and for synchronization, whereas messages are used to supplement information from signals. The whole Management Information System is controlled by a central process which spawns various report generator processes. Signals to the report generators are supplemented by messages which contain information such as process ID's and other data describing actions for the report generators to take. Since many of the report generators have much code and data in common, heavy usage is made of the shared public libraries as supported by the UNIX supervisor. Further functional enhancements are planned for the AEMIS project in the near future. These involve using a process connected to a port for centralized error logging and implementing the status data base manager as a supervisor process.

11.3. Speech Research

The Speech Research Department uses a SEL/86 computer for number crunching applications. The processor is a fast 32-bit machine but it has no mass storage devices of its own. It uses two high-speed inter-computer channels to a PDP-11/45 computer to access file storage. Special kernel and supervisor processes provide the software link between the SEL operating system and the MERT file manager process. The supervisor process communicates with the MERT file manager by means of messages much the same as the UNIX supervisor does. The kernel device driver process controls the channels to the SEL machine. Thus all of the file system primitives available under the MERT and UNIX systems are available in the SEL operating system. Long I/O transfers are used extensively, making use of the contiguous file allocation strategies in the MERT system. The UNIX programming environment in the MERT system is used both for PDP-11 programming and for preparing files and programs to be used in the SEL machine.

11.4. DIR/ECT II

The DIR/ECT II system under development in BIS is to be used by Operating Telephone Companies for the production of White Page Directories and Directory Assistance products. The system will process and validate Operating Company listing information, maintain an on-line listing data base and format and photocompose the directory products. The processing and validation of the input listing data may be characterized as a transaction-oriented subsystem. Listing data, input via either mechanized service orders or interactive terminals, will be handled by a chain of processes that perform functions such as field parsing and interpretation, field validation, context validation and error control. The MERT inter-process communication facilities will be utilized to control the flow of transactions through these modules. Particular use will be made of shared memory for working storage of the transaction data.

As opposed to the relatively short transactions of the input subsystem, the output subsystem for the creation of the directory products may be characterized as a large scale production facility. Here the MERT memory segmentation mechanism is utilized to provide large data segments for in-core layout and mark-up of the photocomposed pages. The primary demand placed on the operating system by the production subsystem will, however, be in the area of extensive data base retrieval operations. To facilitate this requirement, the DIR/ECT II system will have its own listing data base management system. An integral component of the data base manager will be a data base backout and recovery mechanism to maintain the integrity of the data base in the event of a disk or processor failure. The operating system facilities required for the development of such a data base manager include the ability to initiate multiple asynchronous I/O operations, the capability of storing the data contiguously on disk and the ability to receive disk error notification for corrective action by the data base recovery software.

11.5. ESS Feature Loading

In order to do system testing and facility planning, the MERT system is used for loading feature packages on an ESS machine attached to a PDP-11/70. A user program *fpload* is invoked at a remote terminal attached to a #1 ESS test machine. Its function is to allocate storage to user selected components (feature packages) of the ESS program (kept on disk) and then to bind and load the program into ESS memory. The *fpload* user process creates a shareable segment and then creates a supervisor process, *ldc,* which shares the segment. *Fpload* and *ldc* coordinate their activities via events and pass information in the shared segment.

Ldc binds program components and loads the ESS memory. It uses MERT primitives to execute three asynchronous tasks within a single MERT process. The three tasks in effect pipeline the disk file opens, disk reads and binding operations with liberal buffering between tasks.

To load the ESS machine, *ldc* communicates with a kernel character driver, *es*, which handles the hardware interface to the ESS machine. Five megabytes of data on the PDP disk are translated into three megabytes on the ESS side in less than one minute. This corresponds to a transfer rate of 100,000 bytes of data per second from disk. MERT contiguous files and asynchronous physical I/O are required to achieve this throughput.

A hardware tool, *hd* was used to debug the hardware interfaces between the PDP-11/70 and the No. 1 ESS test laboratory. The user process creates a shareable segment and then creates a kernel process which shares this segment. When the user process wishes to read or write virtual kernel space (usually I/O registers in the PDP11), it does so by means of a message to the kernel process. The kernel process can also be attached to a hardware vector and, upon receipt of an interrupt, it will execute a stack of operations in the shared segment.

11.6. RBCS - Record Based Control System

RBCS (10) is a data base management and information report system which has requirements for communication interfaces to support Model 40 CRT's and the need for inter-process communication facilities. The dynamic MERT debugging features provide for quick turnaround in testing new features in a driver. New communication features were added to the TTY40 driver and multi-file capabilities were added to the magtape driver without affecting other interactive users of the system. Changes to the drivers were localized and had no impact on the rest of the system.

At the user level, the programmer can take advantage of shared segments, public libraries, messages and events for a rich set of inter-process primitives. For large amounts of shared data, a shared segment between processes is much more efficient than copying the data through pipes. Common subroutines are placed in a public library to minimize total memory usage and reduce thrashing among processes.

Recently, the RBCS project has built an extended *shell* program (11) which makes use of all of the new system calls which have been added to the MERT/UNIX supervisor. This makes all of these calls available directly to the user at the command level and gives him a rich set of inter-process commands at his fingertips.

11.7. MATS

The Maintenance and Traffic Simulator (MATS) currently in use in the High Capacity Mobile Telephone Lab at Indian Hill runs under the MERT system on a PDP-11/70 computer with only 64K words of memory. MATS consists of three subsystems: the Cell Site Peripheral Simulator (CSPS), the Mobile Call Generator (MCG) and the Monitor. CSPS simulates a full set of cell site peripheral devices (data radios, voice radios, locate radios, mobiles, etc.). MCG monitors and controls real mobile units. Finally, the Monitor displays and records all interactions between the cell site controller (PROCON) and its real peripherals.

The MATS software consists of four kernel-level processes and one user-level program. One kernel process is a driver for the KW11P programmable clock. Another process contains the MATS operating system and device simulators. A third contains the state table Call Controller (CC). The fourth kernel process handles hardware initialization. These four kernel-mode processes and the user program all share two data libraries. The system is a real-time system which requires the kernel dispatcher for fast response and uses common segments for passing data.

12. Protection/Performance Trade-offs

We summarize here the results of our experience with the MERT system as designers, implementers and users. Some of the features added or subtracted from the MERT system have been the result of feedback from various users. We pay particular attention to various aspects of the system design concerning trade-offs made between efficiency and protection. The advantages of the system architecture as well as its disadvantages are discussed.

Each major design decision is discussed with respect to performance versus protection. By protection, we mean protection against inadvertent bugs and the resulting corruption, not protection against security breaches. In general, for the sake of a more efficient system, protection is sacrificed when it was believed that this extra protection would degrade system performance significantly. In most cases the system is used in dedicated applications where some protection could be sacrificed. Maximum protection is provided mainly by separating the various functions into layers, putting each function at the highest possible level, according to the access privileges required. All processes were written in the high-level language, C. This forced some structure in the processes. C controls access to the stack pointer and program counter and automatically saves the general-purpose registers in a subroutine call. This provides some protection which is helpful in confining the access of a program or process.

12.1. Hardware

The hardware of the PDP-11 computers permits a distinction to be made between kernel processes and supervisor processes. Kernel processes have direct access to the kernel-mode address space and may use all privileged instructions. For efficiency reasons, one base register always points to the complete I/O page. This is 4K words of the address space of the PDP-11 computer which is devoted to device addresses. It is not possible to limit access to only the device registers required for a particular device driver. The virtual address space is limited to 16-bit addressing. This presents a limitation to some large processes.

12.2. Kernel

The number of base registers provided by the PDP-11 segmentation unit is a restriction in the kernel. The use of I and D space separation is necessitated to provide a reasonable number (16) of segments. Some degree of protection is provided for the sensitive kernel system tables by the address space separation since the kernel drivers do not use I/D space separation in general. Such kernel processes do not have access to sensitive system data in kernel D-space.

12.3. Kernel Process

Most kernel-mode processes use only kernel I-space. This prohibits access to system segment tables and to kernel code procedures. However access to message buffers, dispatcher control tables and the I/O page is permitted. A kernel process is the most privileged of all processes which the user can load into a running system. The stack used by a kernel process is the same as that used by kernel procedures.

To provide complete security in the kernel would require that each process use its own stack area and that access to all base registers other than those required by the process be turned off. The time to set up a kernel process would become prohibitive. Since kernel processes are most often dispatched to by means of an interrupt, the interrupt overhead would become intolerable, making it more difficult to guarantee real-time response.

In actual practice, the corruption of the kernel by kernel processes occurs very infrequently and then only when debugging a new kernel process. Fatal errors were seldom caused by the modification of data outside of the process's virtual address range. Most errors were timing dependent, errors which would not have been detected even with better protection mechanisms. Hence we conclude that the degree of protection provided for kernel processes in dedicated systems is sufficient without degrading system performance. The only extra overhead for dispatching to a kernel process is that of saving and restoring some base registers and saving the current stack pointer.

12.4. Supervisor Process

Supervisor processes do not have direct access to the segments of other processes, kernel or supervisor. Therefore it is possible to restrict the impact of these processes on the rest of the system by means of careful checking in the kernel procedures. All communication with other processes must go through the kernel. Of course one pays a price for this protection since all supervisor base registers must have the appropriate access permissions set when a supervisor process is scheduled. Message traffic overhead is higher than for kernel processes.

For protection reasons, capabilities were added to the system. This adds extra overhead for each message to the file manager, since each capability must be validated by the file manager. An alternate implementation of capabilities which reduces overhead at the cost of some protection is discussed in a later section.

12.5. Message Buffers

System message buffers are maintained in kernel address space. These buffers are corruptible by a kernel process. The only way to protect against corruption completely would be to make a kernel EMT call to copy the message from the process's virtual address space to the kernel buffer pool. For efficiency reasons this was not done.

For a supervisor process, the copying of a message from the supervisor's address space to the kernel message buffer pool area is necessary. This increases message traffic overhead for supervisor to kernel or supervisor to supervisor transfers. The overhead for sending and receiving a message between kernel processes amounts to 300 usec. whereas for supervisor processes the overhead is of the order of 800 usec. (on a PDP-11/45 computer without cache memory).

12.6. File Manager Process

The file manager process is implemented as a kernel-mode process with I and D space separated to obtain enough virtual address space. In the early implementation stage of the MERT system the file manager was a supervisor process but the heavy traffic to the file manager process induced many context changes and contributed significantly to system overhead. Implementation of the file manager process as a kernel-mode process improved system throughput by an average of about 25 percent. Again this was a protection/efficiency trade-off. Protection is sacrificed since the file manager process has access to all system code and data. In practice, it has not proven to be difficult to limit the access of the file manager to its intended virtual address space. Making the file manager a separate process has made it easy to implement independent processes which communicate with the file manager. The file manager is the only process with knowledge of the detailed structure of the file system. To prevent corruption of the file system, all incoming messages must be carefully validated. This includes careful checking of each capability specified in the message. This is a source of some system overhead which would not exist if the file system was tightly coupled with a supervisor process. However, this separation of function has proven very helpful in implementing new supervisors.

12.7. Process Manager

The process manager is implemented as a swappable supervisor process. Its primary function is to create and start up new processes and handle their termination. An example is the loading of the kernel driver process for the magnetic tape drive. This is an infrequent occurrence, and thus the time penalty to bring in the process manager is tolerable. Other more frequent creations and deletions of processes associated with the UNIX system forking of processes is handled by the system scheduler process. In the early stages of implementation of the MERT system, the creation and deletion of all processes required the intervention of the process manager. This required the loading of the process manager in each case and added significantly to the overhead of creating and deleting of processes.

12.8. Response Comparisons

The fact that a "UNIX-like" environment was implemented as one environment under the MERT kernel, gives us a unique opportunity to compare the overall response of a system running as a general-purpose development system to that of a system running a dedicated UNIX time-sharing system on the same hardware. This gives us a means of determining what system overhead is introduced by using messages as a basic means of inter-process communication.

Application programs which take advantage of the UNIX file system structure give better response in a dedicated UNIX time-sharing system, whereas those which take advantage of the MERT file system structure give a better response under the MERT system. Compute-bound tasks respond in the same time under both systems. It is only where there is substantial system interaction, that the structure of the MERT system introduces extra system overhead, which is not present in a dedicated UNIX system. Comparisons of the amount of time spent in the kernel and supervisor modes using synthetic jobs indicate that the MERT system requires from five to fifty percent more system time for the more heavily used system calls. This translates to an increase of five to ten percent in elapsed time for the completion of a job stream consisting of compilation, assembly, and link-edit. We believe that this overhead is a small price to pay to achieve a well-structured operating system with the ability to support customized applications. The structure of the system provides a basis for doing further operating system research.

13. Design Decisions in Retrospective

A number of design decisions were made in the MERT system which had no major impact on efficiency or protection. However, many of these impacted the interface presented to the user of the system. The pros and cons of these decisions are discussed here.

13.1. File System

The first file system for the MERT system was designed for real-time applications. For that, it is well-suited. For those applications which require the collection of data at a high rate, the use of contiguous files and asynchronous I/O proved quite adequate. However, the number of applications which required contiguous files was not overwhelming. For those applications which used the MERT system as a development system as well, the allocation of files by extents is not optimal, although adequate. The number of files which exhausted their 27 extents was small indeed. Also the need for compaction of file systems due to fragmentation was not as great as might have been expected and seems not to have posed any problems. The root file system very rarely needs to be compacted due to the nature of file system activity on it.

The file manager process uses multi-tasking to increase its throughput. This has added another degree of parallelism to the system, but on the other hand has also been the source of many hard-to-find timing problems.

The use of 16-bit block numbers is a short-coming in the file system with the advent of larger and larger disks. However, this has been rectified in a new 32-bit file system which has features that make it more suitable for small time-sharing files and yet allows the allocation of large contiguous files. Compaction of this file system is not required.

13.2. Error Logging

A special port process to collect error messages has proven to be very useful for tracking down problems with the peripheral devices. Sending messages rather than printing diagnostics out at the control terminal minimizes impact on real-time response. One drawback of this means of reporting errors is that the user is not told of the occurrence of an error immediately at his terminal unless the error is unrecoverable. He must examine the error logger file for actual error indications.

13.3. Process Ports

Process ports were implemented as a means of enabling communication among unrelated processes. This has proven to be an easy-to-use mechanism for functions such as the error logger. Other uses have been made of it such as a centralized data base manager. The nature of the implementation of ports requires that the port numbers be assigned by some convention agreed upon by users of ports. Probably a better implementation of ports would have been to use named ports, i.e. refer to ports by name rather than by number. The number then is not dependent on any user assigned scheme.

13.4. Shared Memory

Shared memory allows the access to a common piece of memory by more than one process. The use of named segments to implement sharing enables two or more processes to pass a large amount of data between them without actually copying any of the data. The PDP-11 memory management unit and the 16-bit virtual address space are limitations imposed on shared memory. Only up to 16 segments may be in a process' address space at any one time. Sometimes it would be desirable to limit access to less than a total logical segment. The implementation chosen in the MERT system does not allow this.

13.5. Public Libraries

Public libraries are used in the MERT system at all levels, kernel, supervisor and user. The use of public libraries at the kernel level has allowed device drivers to share a common set of routines. At the user level, many programs have made use of public libraries to make a substantial savings in total memory requirements. The initial implementation of public libraries required that when a public library was reformed, all programs which referenced it had to be link-edited again to make the appropriate connection to subroutine entry points in the public library. The current implementation makes use of transfer vectors at the beginning of the public library through which subroutine transfers are performed. Thus if no new entry points are added when a public library is formed again, the link-edit of all programs which use it becomes unnecessary. This has proven to be very helpful for maintaining a set of user programs which share public libraries. It has proven to be convenient also for making minor changes to the system library when new subroutines are not added. This makes the re-forming of all device drivers unnecessary each time a minor change is made to a system library.

13.6. Real-Time Capabilities

The real-time capabilities of the MERT system are determined in part by the mode of the process running, i.e. kernel or supervisor. As a kernel-mode process, the process is dispatched to by an interrupt or an event. Time-out events may be used effectively to guarantee repetitive scheduling of a process. The response of a kernel process is limited by the occurrence of high priority interrupts, and therefore can only be guaranteed for the highest priority process. A supervisor process' scheduling priority can be made high by making it a non-swap process and giving it a high software priority. A response of the order of a few milliseconds can then be obtained. The scheduler uses pre-emption to achieve this. One aspect missing from the scheduler is dead-line scheduling. Thus it cannot be guaranteed that a task will finish by a certain time. The requirement for pre-emption has added another degree of complexity to the scheduler and of necessity adds overhead in dispatching to a process. Pre-emption has also complicated the handling of critical regions. It is necessary to raise the hardware priority around a critical region. This is difficult to do in a supervisor since it requires making a kernel EMT call, adding to response time. Shifting of segments in memory also adds to the response time which can be guaranteed.

13.7. Debugging Features

Overall, the debugging features provided by the MERT system have proven to be adequate. The kernel debugger has proven useful in looking at the history of events in the kernel and examining the detailed state of the system both after a crash and while the system is running. In retrospect it would have been helpful to have some more tools in this area to examine structures according to named elements rather than by offsets.

The dynamic loading, dumping and then debugging of processes, both kernel and supervisor on a running system have been helpful in achieving fast debugging turn-around. While post mortem debugging is useful, interactive debugging would eliminate the need to introduce traces and local event logging to supervisor and kernel processes as debugging aids. One danger of planting break-point traps at arbitrary points in the UNIX supervisor has been that of planting them in a critical region in which a resource is allocated. The resource may not be freed up properly and other processes may hang waiting for the resource to be freed up.

13.8. Memory Manager

The memory manager is a separate kernel process and handles incoming requests as messages in a fairly sequential manner. One thing it does do in parallel, however, is the loading of the next process to be run while the current one is running. In certain cases the memory manager can act as a bottleneck in the system throughput. This can also have serious impact on real-time response in a heavily-loaded system.

13.9. Scheduler

The scheduler in the MERT system is another separate kernel process. One improvement which could be made in this area is to separate mechanism from policy. The fact that the scheduler and memory manager are separate processes has system-wide impact in that the scheduler cannot always tell which process is the best one to run based on which one has the most segments in memory. The memory manager does not tend to throw out segments based on which process owns it but rather on usage statistics.

13.10. Messages

Messages have proven to be an effective means of communication between processes. At the lowest level, they have been helpful in separating functions into processes and of making these processes modular and independent. It has made things like error logging easy to implement. Communication with the file manager process by means of messages has removed the dependency of supervisor processes on file system structures. In fact a number of different file managers have been written to run using the identical "UNIX-like" supervisor. The UNIX file manager was brought up to run in place of the original MERT file manager without any impact on the supervisor processes. Messages at a higher level have not always been easy to deal with. It is difficult to prevent a number of user processes from swamping the kernel message buffer pool and thereby impacting system response.

The MERT system implementation of messages solves the problem of many processes sending to one process quite effectively. However, the reverse problem of one process sending to many processes (i.e. many servers) is not handled efficiently at all.

13.11. Firewalls

Having separate processes for separate functions has modularized the design of the system. It has eased the writing of new processes but required them to obey a new set of rules. To ensure that processes obey these rules requires an amount of checking which would not be necessary if processes were merged in one address space. This has been especially true of the file manager where corruption of data is very crucial as it can very quickly spread as a cancer in the system.

14. Portability

Recently a great deal of interest has been expressed in porting complete operating systems and associated user programs to hardware configurations other than the DEC 16-bit PDP-11 computer. We discuss here some of the hardware characteristics on which the MERT system depends and the impact of these on the software.

14.1. Crucial Hardware Aspects

At the time that we designed the MERT operating system (circa 1973), the DEC PDP-11/45 processor with a memory management unit allowing the addressing of up to 124K words of memory was a new system. Moreover, the memory management unit was rather sophisticated for mini-computers at that time since it supported three address modes: kernel, supervisor and user. It also supported two address spaces per mode, instruction and data. This enables a mode to address up to 64K words in its address space. Two address modes are generally sufficient for operating systems which provide one environment to the user. To support multi-environments, three modes are required (or at least are desirable), one of which provides the various environments to the user. We decided to make use of this feature. The separation of instruction and data address space provides more address space for a process. It also provides a greater number of segments per user and allows some degree of protection. This was used in the kernel where a large number of separate pieces of code and data need to be referenced concurrently. The protection provided is made use of in kernel processes which need very few base registers and do not need access to very much data, in fact the less the better. Thus a kernel process is not allowed to run with instruction and data space separated so as to protect sensitive system tables.

The third unique feature of the PDP-11/45 computer is that it has a programmable interrupt register (PIR). This enables the system to trigger a software interrupt at one of seven hardware priority levels. The interrupt goes off when the processor starts to run at less than the specified priority. This is used heavily in the MERT system scheduler process and by kernel system routines which trigger various events to occur at specified hardware priorities. It is not sufficient to depend on the line clock for a pre-emptive scheduler to guarantee real-time response.

We have identified here three unique features of the PDP-11/45 processor (and the PDP-11/70) which have been heavily used in the MERT system. These features are identified as unique in that a general class of mini-computers do not have all of these features, although some may have one or more. They are also identified as unique in that the UNIX operating system has not made critical use of them. Therefore the portability of the UNIX system is not impacted by them. We discuss in the next section some of the crucial problems which must be solved in porting the MERT system to other hardware architectures.

14.2. Portability Considerations

The first problem we address is that of a common language. Currently most of the MERT system is written in C. This includes all device driver processes, the scheduler, the file manager, the process manager and the UNIX supervisor. Most of the basic kernel, including the memory manager process, is written in PDP-11 assembly language. This portion is of course not portable to other machines. Recently, a portable C compiler (12) has been written for various machines, both 16-bit and 32-bit machines, the two classes of mini-computers which are of general interest for portability purposes. These include the PDP-11, the Interdata 8/32 and the 3B machine (32-bit machine). It is these machines which we discuss here.

The UNIX system has been ported to the Interdata 8/32 machine, this includes all user programs as well as the operating system itself (13). Thus if the portability of the MERT system to the Interdata 32-bit machine were to be considered, all user programs have already been ported. The main pieces of software which have to be written in portable format include all device drivers, the scheduler, the process manager, the file manager and the UNIX supervisor. Of these only the device drivers have machine dependencies and need substantial rewriting. The file manager, being a kernel process has some machine dependent code. The bulk of the software which must be rewritten is in the kernel itself, being substantially written in PDP-11 assembly language. Also all library interface routines must be rewritten. Many of the calling sequences for library routines have to be reworked since arguments are passed specifically as 16-bit integers. Some sizes, especially of segments, are specified in terms of 16-bit words. For portability reasons, all sizes must be treated in terms of 8-bit bytes.

The second major consideration in the portability of the MERT system is the hardware as mentioned in the previous section. A portability effort is currently under way with the 3B hardware. This is a 32-bit machine; hence the software issues raised above become very real. The memory management unit for the processor is sufficiently powerful that the MERT system can be supported. A C compiler for the 3B machine exists. The machine supports multiple address spaces, sufficient to cover the three required to support the MERT system. The number of possible segments are more than enough for all of the requirements of the current MERT system. The 3B processor has 16 hardware priority levels and has the equivalent of a programmed interrupt register. Thus it has all of the features required to support a MERT system.

15. Reflections

In designing any system, one must make a number of crucial decisions and abide by them in order to come up with a complete and workable system. We have made a number of these, some of which have been enumerated and discussed in the above sections. Upon reflecting on the results and getting feedback from users of the MERT system, we have come up with a number of design decisions which could probably have been made differently from what was actually done. Users have pushed the system in directions which we never considered, finding holes in the design and also some bugs which were never exercised in the original MERT system.

15.1. Capabilities

Capabilities were implemented in the system as a result of the experience of one user in writing a new supervisor process which sent messages to the file manager. There were two major deficiencies. The first had to do with protection. Under the old design (without capabilities) it was possible to ignore the protection bits. Upon reading/writing a file, no check was made of the protection bits. As long as a file was open, any action could be taken on the file, reading or writing; this included directories. With the addition of capabilities, when a file is opened, the capability is put in the user's PCB. The capability includes the entry number in the file manager tables, protection bits and a usage count. The capability is put in a message to the file manager by the kernel when a request is made to read/write a file. These three quantities are checked by the file manager. A capability must be satisfactorily validated before an access can be made to a file. This provides the degree of protection desired.

The second deficiency of the file manager had to do with the maintenance of an up-to-date set of open file tables. If a process is abnormally terminated, i.e. terminated by the scheduler without being given a chance to clean up, the process may not have been able to close all its files. This would typically occur when a breakpoint trap was planted in an experimental version of the UNIX supervisor. The fact that no table is maintained in a central place with a list of all files open by each process caused file tables to get out of synchronization. Capabilities provide such a central table to the process manager and the memory manager. Thus when an abnormal termination is triggered on a process, the memory manager can access the process PCB and take down the capabilities one by one, going through the capability list in the PCB, sending close messages to the file manager. This provides a clean technique for maintaining open counts on files in the file manager tables.

In retrospect, the implementation of capabilities in the MERT system was probably carried to an extreme, i.e. not in keeping with the other protection/efficiency trade-offs made. The trade-off was made in favor of protection rather than efficiency, in this case. The current implementation of capabilities is expensive in that extra messages are generated in opening and closing files. For instance, in closing a file, a *close* message is sent to the file manager; this in turn generates a message to the capability manager (i.e the memory manager) to take down the capability from the PCB of the process which sent the *close* message. The asynchronous message is necessary since the memory manager process must bring the PCB into memory to take down the capability if the PCB is not already in memory.

A more efficient means of achieving the same result would be to maintain this list of capabilities in the supervisor address space with general read/write permissions with a pointer to the capability list maintained in the PCB. It would then be the supervisor's responsibility to fill in the capability when sending a message to the file manager and to take down the capability when closing the file. This requires no extra message traffic overhead as compared to the original implementation without capabilities. Upon abnormal termination, the memory manager could still go through the capability list to take down all capabilities by sending *close* file messages to the file manager. Protection is still achieved by the encoded capability. Efficiency is maintained by eliminating extra messages to the memory manager. This proposed implementation also has the added benefit that it can be implemented for kernel processes in the same manner, i.e. using a pointer to a capability list in the kernel process header.

15.2. Global System Buffers

In the current implementation of the MERT system, each process maintains its own set of system buffers. The file manager provides its own set of buffers, used entirely for file mapping functions (e.g. superblocks for mounted file systems, inodes and directories). The UNIX supervisor provides its own set of buffers for use by all UNIX processes. These buffers are used almost exclusively for the contents of files. However, it is possible for a file to be the image of a complete file system, in which case a buffer may actually contain the contents of a directory or inode. This means there may be more than one copy of a given disk block in memory simultaneously. Because of the orthogonal nature of the uses of buffers in the UNIX system and the file manager, this duplication hardly ever occurs and does not pose a serious problem. Within the UNIX system itself all buffers are shared in a common data segment.

However, if one wishes to implement other supervisors and these supervisor processes share a common file system with the UNIX supervisor, it becomes quite possible that more than one copy of some disk blocks exist in memory. This presents a problem for concurrent updates.

An alternate method of implementation of buffers would have been to make use of named segments to map buffers into a globally accessible buffer pool. The allocation and deallocation of buffers would then become a kernel function and this would guarantee that each disk block would have a unique copy in memory. If the MERT system had allowed protection on sections of a segment, then system buffers could have been implemented as one big buffer segment broken up into protectable 512-byte sections. The system overhead in this implementation probably would have been no greater than the current implementation. Each time a buffer is allocated and released a kernel EMT call would be necessary. However, even the present implementation requires two short-duration EMT calls to prevent process pre-emption during a critical region in the UNIX supervisor both during the allocation and releasing of a buffer.

15.3. Diagnostics

One of the shortcomings of the MERT system has been the lack of system diagnostics printed out at the control console reporting system troubles. The UNIX system provides diagnostic print-outs at the control console upon detection of system inconsistencies or the exhaustion of crucial system resources such as file table entries, inode table entries or disk free blocks. Device errors are also reported at the control console. In the MERT system, device errors are permanently recorded on an error logger file. One reason for not providing diagnostic print-out at the control console is that the print-out impacts real-time response.

The lack of diagnostic messages has been particularly noticeable in the file system manager and in the basic kernel when resources are exhausted. Providing diagnostic messages in the system requires the use of some address space in each process making use of diagnostic messages; this would require duplication of the basic printing routines in the kernel, the file manager and any other process which wished to report diagnostics or the inclusion of the printing routines in the system library. A possible solution would have been to make use of the MERT message facilities to send diagnostic data to a central process connected to a port to print out all diagnostics both on the control console and into a file for later analysis. Using this technique, it would also be possible to send diagnostic messages directly to the user's terminal which caused the diagnostic condition to occur. The diagnostic logger process would be analogous to the error logger process.

15.4. Scheduler Process

The current MERT scheduler is a separate kernel system process which implements both the mechanism and the policy of system-wide scheduling. It would be more flexible to implement only the mechanism in the kernel process and let the policy be separated from this mechanism in other user-written processes.

Acknowledgments

Some of the concepts incorporated in the kernel were developed in a previous design and implementation of an operating system kernel by Mr. C. S. Roberts and one of the authors (H. Lycklama). The authors are pleased to acknowledge Mr. C. S. Roberts for many fruitful discussions during the design stage of the current operating system.

We are grateful to the first users of the MERT system for their initial support and encouragement. These include: S. L. Arnold, W. A. Burnette, L. L. Hamilton, J. E. Laur, J. J. Molinelli, R. W. Peterson, M. A. Pilla and T. F. Tabloski. They made suggestions for additions and improvements, many of which were incorporated into the MERT system, and make the MERT system what it is today.

The current MERT system has benefited substantially from the documentation and debugging efforts of E. A. Loikits, G. W. R. Luderer and T. M. Raleigh.

H.L.

H. Lycklama

DLB

D. L. Bayer

HO-3114-HL-troff
MH-1352-DLB-troff

Attachments:
  References
  Figures 1-4

References

(1) Dijkstra, E.W., The Structure of the 'THE' Multi-Programming System. Comm. ACM 11, (May 1968), p341.

(2) Brinch Hansen, P., The Nucleus of a Multi-Programming System. Comm. ACM 13, (April 1970), p238.

(3) Sorenson, P. G., Interprocess Communication in Real-Time Systems, Proc. Fourth ACM SOSP, Oct. 1973, pp 1-7.

(4) PDP-11/45 Processor Handbook, Digital Equipment Corporation, Maynard MA, 1971.

(5) Wulf, W. A., HYDRA - A Kernel protection system, Proc. AFIPS 43, NCC pp 998-999, 1974.

(6) Frailey D. J., DSOS - A Skeletal, Real-Time, Minicomputer Operating System, Software - Practice and Experience, 5, pp 5 - 18, 1975.

(7) Sevcik, K.C., Atwood, J.W., Grushcow, M.S., Holt, R.C., Horning, J.J. and Tsichritzis, D. Project SUE as a Learning Experience. Proc AFIPS 41, Pt. 1, FJCC pp. 331-339, 1972.

(8) Thompson, K. and Ritchie, D.M., The UNIX Time-Sharing System. Comm. ACM 17, (July 1974), p365.

(9) Ritchie, D. M., C Reference Manual, TM-74-1273-1.

(10) Pilla, M. A., CORDS/RBCS under the MERT System, Private Communication.

(11) Bourne, S., The Command Interpreter in the UNIX System, Private Communication.

(12) Johnson, S. C., Portable C Compiler, Private Communication.

(13) Johnson, S. C. and Ritchie, D. M., Portability of the UNIX System, Private Communication.

SYSTEM STRUCTURE

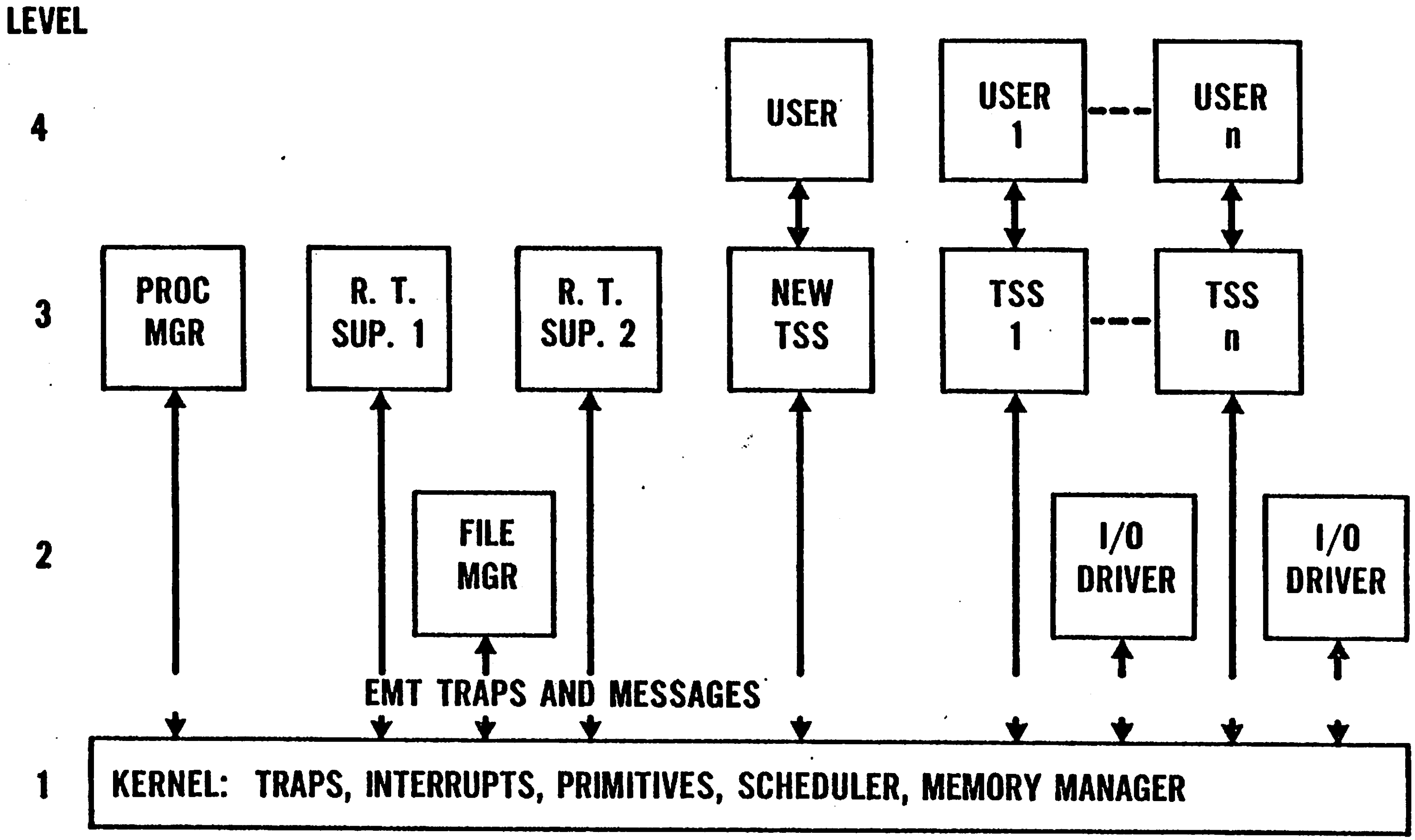
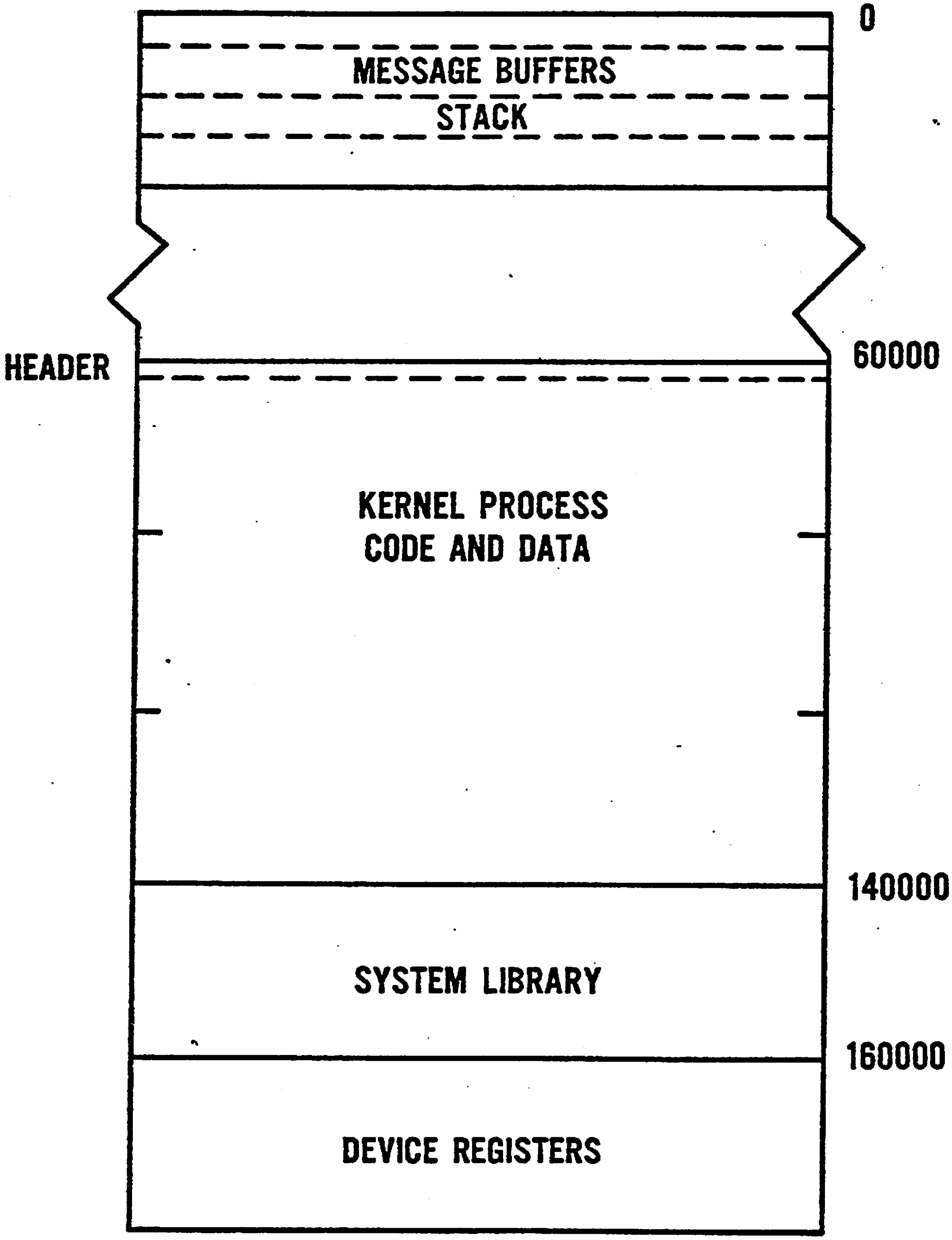
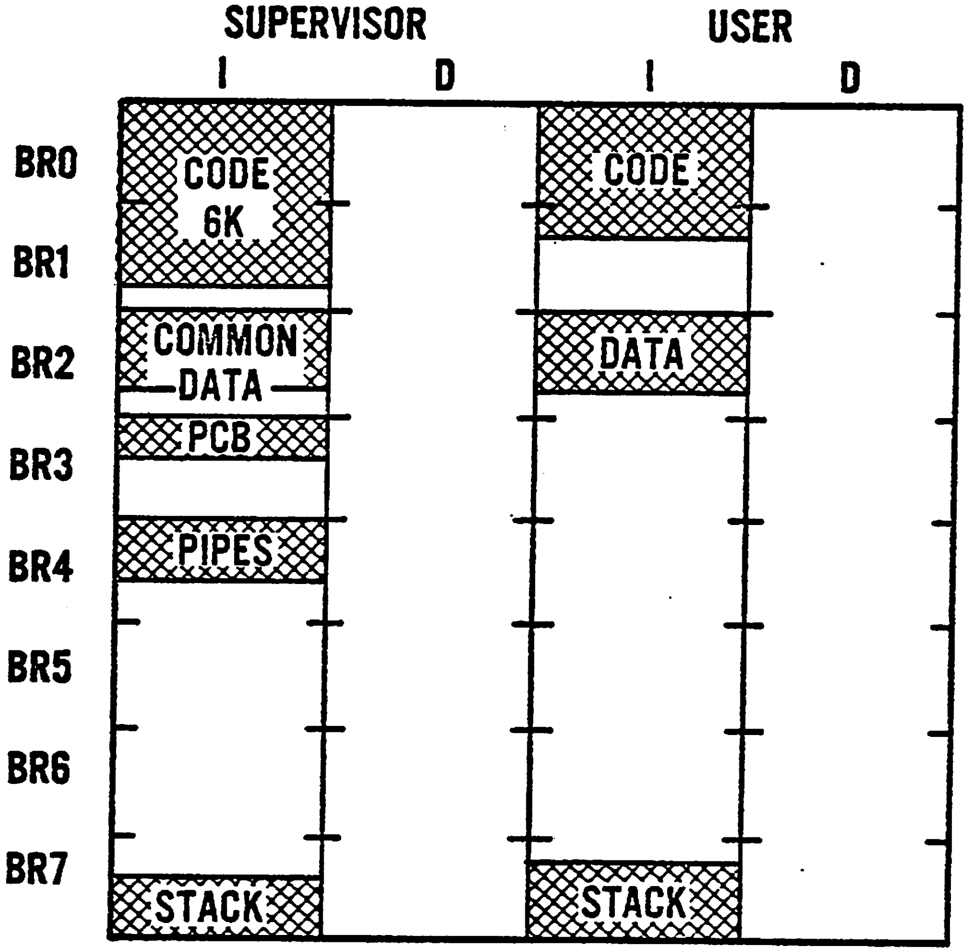


FIGURE 1



THE VIRTUAL ADDRESS SPACE OF A TYPICAL KERNEL PROCESS

FIGURE 2



UNIX PROCESS VIRTUAL
ADDRESS SPACE

FIGURE 3

<table>
<tr><td colspan="2">LINK POINTER</td></tr>
<tr><td colspan="2">FROM PROCESS NUMBER</td></tr>
<tr><td colspan="2">TO PROCESS NUMBER</td></tr>
<tr><td>TYPE</td><td>SIZE</td></tr>
<tr><td colspan="2">IDENTIFIER</td></tr>
<tr><td>SEQUENCE NUMBER</td><td>STATUS</td></tr>
<tr><td colspan="2">MESSAGE
DATA</td></tr>
</table>

MESSAGE FORMAT

FIGURE 4