MERT - A Multi-Environment Real-Time Operating System

D. L. Bayer
H. Lycklama
Bell Laboratories
Murray Hill, New Jersey 07974

MERT is a multi-environment real-time operating system for the Digital Equipment PDP-11/45 and 11/70 computers. It is a structured operating system built on top of a kernel which provides the basic services such as memory management, process scheduling, and trap handling needed to build various operating system environments. Real-time response to processes is achieved by means of preemptive priority scheduling. The file system structure is optimized for real-time response. Processes are built as modular entities with data structures that are independent of all other processes. Interprocess communication is achieved by means of messages, event flags, shared segments, and shared files. Process ports are used for communication between unrelated processes.

Keywords: kernel, operating system, process, supervisor.

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 principals of structured modularity (1),(2) is the correct approach to building an operating system. The influence of a process must be confined to an environment which is well protected from the rest of the system and must never affect the state of other environments.

MERT is an executive which provides an environment which is more conducive for the implementation of operating systems than a raw machine. The executive 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 MERT and define the services available to user programs. The operating systems are independent. Communication and synchronization primitives and shared memory permit varying degrees of co-operation between independent operating systems.

The MERT system runs on the DEC PDP-11/45 and PDP-11/70 computers (3). These computers provide an eight-level hierarchical 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.

The basic computer hardware resources consist of the actual memory, the CPU and the various I/O devices. The first level (see Fig. 1) of the operating system structure, called the kernel, controls and allocates these resources. The kernel consists of a set of highly privileged procedures and therefore must be very reliable.

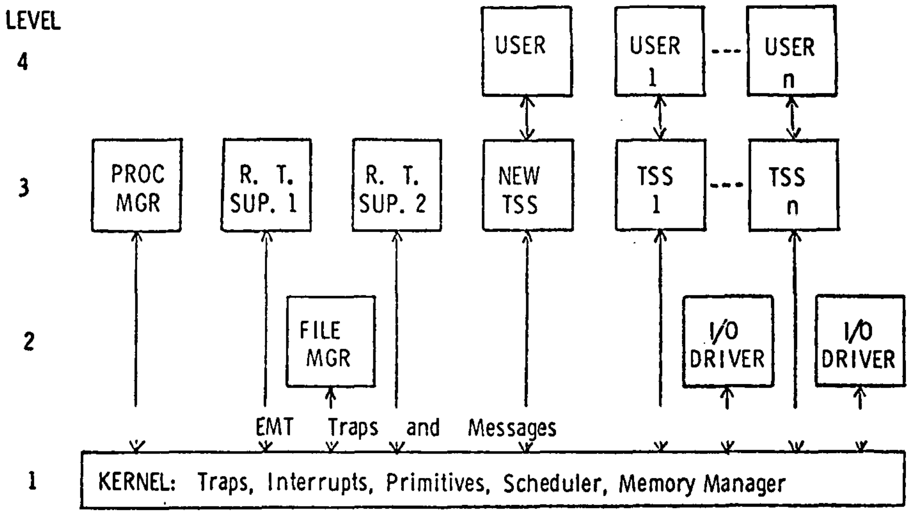

Figure 1 System Structure

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.

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.

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 catch the user traps. Actually the user procedure is merely an extension of the supervisor process. This is the highest level of protection provided by the computer hardware.

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 (4). 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.

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. Some trade-offs are given that have been made for efficiency reasons thereby sacrificing some protection. Some operational statistics are also included here.

2. Segments

We define a logical segment as 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 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.

3. Processes

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

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 (segmentation 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 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) (segmentation 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 segmentation 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.

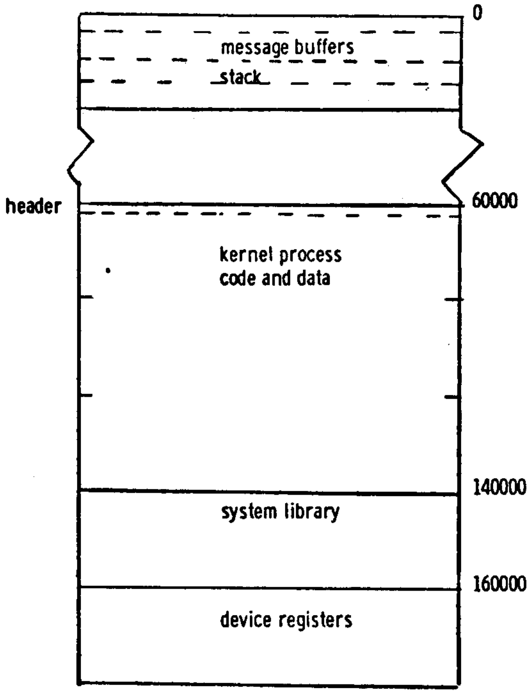

Figure 2 The virtual address space of a typical kernel process

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.

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 consists of 32K words of instruction space and 32K words of data space in both supervisor and user modes. Of this 128K, at least part of each of three segmentation registers (12K) must be used for access to:

1) the process control block, a segment typically 128 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.

2) the process supervisor stack and data segment.
3) the read-only code segment of the supervisor.

The rest of the address space is controlled by the supervisor through EMT traps to the kernel.

4. The Kernel

The concept of an operating system nucleus or kernel has been used in several systems. However, each system has included a different set of logical functions (5), (6). In this section the logical structure of the MERT kernel is discussed. The modules which are considered part of the nucleus are distinguished from the memory management and scheduler processes. A discussion of the scheduling policy is also given.

4.1 Kernel Modules and Processes

The kernel consists of a process dispatcher, a trap handler, and routines (procedures) which implement the system primitives. Approximately 5.5K 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 those 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 I/O. 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 road blocking, changing the scheduling priority, or making the segments of the process nonswap or swappable.
8) Miscellaneous services such as reading the console switches.

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

The memory manager process communicates with the rest of the system via messages and is capable of handling three 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.

The scheduler process 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.

4.2 Dispatching and Scheduling Policy

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.

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 is higher priority than the current process, the new process will preempt the current process. A pointer to the preempted process is saved in the process control block of the preempting process. This pointer is used to return control to the interrupted process.

To minimize thrashing and swapping, the scheduler uses a "will receive an event soon" flag which is set by the process when it road blocks. This flag is typically set when a process road blocks awaiting completion of I/O which is expected to finish in a short time relative the length of a 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 times out.

5. Inter-Process Communication

A structured system requires a well-defined set of communication primitives to achieve 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.

5.1 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.

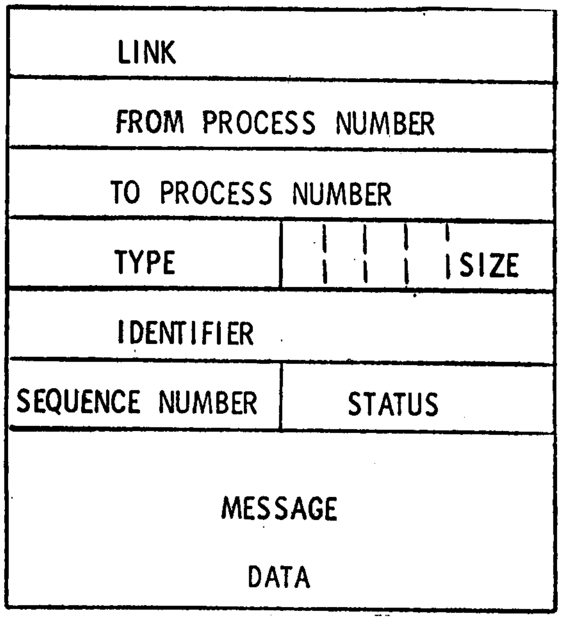

Figure 3 Message Format

5.2 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 3. 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.

5.3 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.

5.4 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 a later section), 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 his 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.

5.5 Files

The file system has a hierarchical structure equivalent to the UNIX file system (8) and as such has certain protection keys (see section 6). Most files have general read/write permissions and the contents are shareable between processes.

In some cases the access permissions of the file may itself 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.

5.6 Process Ports

Knowing the identity of a process gives another 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 to occur. 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 6). 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 in the MERT system. A process port is a globally known "device" 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.

6. 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 real-time applications.

For data base management systems, files must 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 induce 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 for that file. A file may have more than one link to it, thus enabling the sharing of files.

Special files in MERT 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 bitmap which resides on secondary storage. If the in-core free list should become exhausted, the bitmap 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.

7. A Time-Sharing Supervisor

One of the first supervisor processes 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 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 re-entered. Note that a simple semaphore cannot prevent such re-entry 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 6000 words. All memory management and process scheduling functions are performed by the kernel.

8. 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(4).

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.

9. Process Debugging

One of the most powerful 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 inter-processor 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.

10. Summary

We summarize here some of the conclusions we have come to concerning the structure of the system, its overall efficiency, the design trade-offs made, the disadvantages of the system design as well as the advantages and some operational statistics. In general, for the sake of a more efficient system, protection was sacrificed where it was believed not to be crucial to an effective system. The very nature of the structure of the C language which was used to write the code for all processes, kernel and supervisor, forced structure in the processes thus providing some means of protection.

The hardware of the PDP-11/45 and PDP-11/70 computers requires that a distinction 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. Moreover, a kernel process has access to some of the sensitive system data used by the kernel procedures. The stack used by a kernel process is the same as that used by the basic kernel. The address sharing expedites the transmission of messages since the data in the message need not be copied.

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 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.

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

In actual practice the corruption of the kernel by kernel processes does not occur in our system even when debugging new kernel processes. Using the C language facilitated the writing of correct program procedures. We observed that even in the debugging stage fatal system errors were never caused by the modification of data outside of a process's virtual address range. Most errors were timing dependent, errors which would not have been detected even with better protection mechanisms.

Supervisor processes do not have direct access to segments of other processes, kernel or supervisor. Therefore it is possible to restrict the effect of these processes on other processes. Of course one pays a price for this protection in the sense that all supervisor base registers must have the appropriate access permissions set when the process is scheduled. Message traffic overhead is also higher now because a sendmsg kernel primitive must copy the message from the process's virtual address space to the system message buffer. Similarly a getmsg kernel primitive must copy the message from the kernel message buffer to the process's virtual address space. The following times are indicative of the system overhead involved in sending and receiving messages:

	kernel	supervisor	
send	150	400	usec.
receive	150	400	usec.

The total system design gives us a unique opportunity to compare system response time running under a dedicated UNIX time-sharing system with the response time running in a UNIX time-sharing environment supported by the MERT system. 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 MERT. Compute-bound tasks of course 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. Heavily used programs typically take 5 to 10 percent longer to run under MERT compared to dedicated UNIX at the current stage of implementation. We believe that this overhead is a small price to pay to achieve a well-structured operating system which has capabilities for further expansion in supporting other processes which provide different environments. In retrospect we believe the structure of the system does provide a good base for doing further operating system research.

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.

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

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

(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. Internal memorandum, Bell Telephone Laboratories (1973).