



The information contained herein is for the use of employees of Bell Laboratories and is not for publication (see GEI 13.9-)

Title: The MERT/UNIX* Supervisor

Date: April 20, 1978

Other Keywords: Time Sharing
Real Time
Interprocess Communication
Asynchronous I/O

TM: 78-3114-4

Author(s)
H. Lycklama

Location
HO 1G-317

Extension
3212

Charging Case: 39394
Filing Case: 39394-11

ABSTRACT

A UNIX-like supervisor was implemented as an environment in the MERT system. It provides essentially all of the capabilities available to a user program running under the UNIX operating system but in addition offers a number of other capabilities unique to the MERT/UNIX supervisor. These enhancements use the capabilities of the MERT kernel and the structure of the MERT file system and include the ability to:

- create a new environment
- send and receive messages
- send and receive events
- set up shared segments
- manipulate contiguous files
- set up and communicate with process ports
- initiate physical and asynchronous I/O.

This paper describes these capabilities as well as their implementation. Some typical uses of the MERT/UNIX facilities are discussed.

*

UNIX is a trademark of Bell Laboratories

Pages Text: 13	Other: 3	Total: 16
No. Figures: 0	No. Tables: 0	No. Refs.: 10



Bell Laboratories

subject: **MERT/UNIX* Supervisor**
Case: **39394**
File: **39394-11**

date: **April 20, 1978**

from: **H. Lycklama**
HO 3114
1G-317 x3212

TM: **78-3114-4**

ABSTRACT

A UNIX-like supervisor was implemented as an environment in the MERT system. It provides essentially all of the capabilities available to a user program running under the UNIX operating system but in addition offers a number of other capabilities unique to the MERT/UNIX supervisor. These enhancements use the capabilities of the MERT kernel and the structure of the MERT file system and include the ability to:

- create a new environment
- send and receive messages
- send and receive events
- set up shared segments
- manipulate contiguous files
- set up and communicate with process ports
- initiate physical and asynchronous I/O.

This paper describes these capabilities as well as their implementation. Some typical uses of the MERT/UNIX facilities are discussed.

MEMORANDUM FOR FILE

1. Introduction

The MERT/UNIX process was the first supervisor process developed for the MERT system. The general design and implementation details of the MERT system have been covered in previous papers [1,2,3]. The UNIX time-sharing system itself is described elsewhere [4] as well as in the UNIX Programmer's Manual [5]. The MERT/UNIX supervisor is logically equivalent to the UNIX system in that it implements all of the system calls of the UNIX system. A user program running under the MERT/UNIX supervisor sees no difference from a program running under the UNIX system in a functional sense.

The MERT/UNIX supervisor supports a number of additional system calls to take advantage of the features provided by the MERT kernel and the MERT file system (see Appendix A). This

* UNIX is a Trademark of Bell Laboratories.

memorandum describes the implementation of the MERT/UNIX supervisor and gives a brief overview of the new system entry points added to give the reader an appreciation of how the new features may be used. The final section of this memorandum gives examples of how some of the features have been used by various utility programs and some projects. References to previous publications are provided where appropriate.

2. Creating New Environments

There are a number of reasons why one may wish to establish an environment different from the UNIX time-sharing system environment under which to run. These include the ability to:

- test out a new process, either supervisor or kernel.
- establish shared memory between a kernel process and a supervisor/user process.
- run supervisor process in supervisor mode only, to maintain close control over the process and reduce overhead.
- run new environment, e.g. DEC RSX11 system.
- run kernel process.

One system call in the MERT/UNIX supervisor is provided to perform this function:

```
pid = pcreat("pfile", flag)
```

The file *pfile* contains an image of the process to be loaded. This file must previously have been created by the *ldp* utility command [6]. The header of the file (first 512 bytes) contains information as to which segments make up the process, the mode of the process, whether public libraries are to be included, etc. If the created process wishes to share a segment with its parent process, the parent process must create the segment with permissions "07" so that it will be inherited. Only one segment can currently be shared this way. The second argument *flag* indicates whether or not the process waits for a message to start it off when first created. *Pcreat* sends a message to the process manager to create the process. The process identifier of the child process is returned in *pid*.

3. File System Primitives

The MERT file system structure is hierarchical as it is in the UNIX system. The MERT file system supports the same type of files as the standard UNIX file system plus a few different "MERT-type" files. The types of files supported include directories, ordinary, special and contiguous. The contiguous file type is unique to the MERT system. This file type is made possible by the allocation of a number of consecutive blocks to a file. In the UNIX file system, one block is allocated to a file at a time. In the MERT file system, blocks are allocated by extents, i.e. starting block number and number of consecutive blocks. One file may have up to 27 extents. A file marked as contiguous can have only one extent by definition. A MERT file system with more flexibility is discussed in a companion paper [7].

Two system calls are provided in the MERT/UNIX supervisor to deal with contiguous files. The system call:

```
desc = falloc(fname, mode, size0, size1)
```

is used to create a new contiguous file *fname* with mode bits *mode* and size, in bytes, given by the 32-bit value in *size0* and *size1*. Space is pre-allocated for this file and it may not be grown. Similar to the standard UNIX system call *creat*, the file descriptor is returned in *desc*. One may now write at random into the file. An attempt to write beyond the end of the file will fail since only one extent may be used for this contiguous file. The file system space allocated to this file is only returned to the free list when the last link to the file is removed, regardless of how much space is used. Large physical and asynchronous I/O transfers may be done on this file with the assurance of good real-time response.

Occasionally, a file may use up all of its 27 extents and cannot grow any further, or a file to/from which one needs optimum real-time response may actually consist of several extents. The system call:

```
fmove(dev, inode, flag)
```

can be used to move all of the file extents into one large contiguous extent. Although only one extent

is used, this file is not marked contiguous and thus may still grow in size if necessary. The arguments, *dev* and *inode*, are obtained from a previous call to the *fstat* system call [5]. The file is locked and cannot be used while it is being moved. The file is moved into a contiguous area of the file system starting at a lower block address if possible. If the value of *flag* is non-zero and the file cannot be moved into a contiguous area lower than that in which it currently resides, it is moved into the first available contiguous space (i.e. lowest block number available).

4. Physical and Asynchronous I/O

The normal mode of data transfer to/from a device for all files is system buffered I/O. In the UNIX system, it is possible to do physical (raw) I/O directly to/from a program's data area only for special files such as block device files. These mechanisms have been enhanced in the MERT/UNIX supervisor to allow a very flexible set of I/O capabilities. An additional file type, a record-type file, has also been added to deal with record-oriented devices such as magtape in a more flexible manner.

In MERT/UNIX, three types of data transfers are available to a user program. These are referred to as buffered, physical and asynchronous. The following table summarizes the types of data transfers possible for each of the file types:

File Types	Buffered	Physical	Asynchronous
Ordinary	x	x	x
Directory	x	x	x
Contiguous	x	x	x
Block	x	x	x
Record	x	x	x
Character	x	-	-

A system call:

`setio(desc, iomode)`

is provided to set the I/O mode of all subsequent read/write requests for the file defined by the file descriptor *desc*. The I/O modes are:

iomode	type
0	buffered
1	physical
2	asynchronous

Buffered I/O is the normal mode of data transfer in the UNIX/MERT system and involves the buffering of data in the system before it is transferred to the user's address space for read operations and after it is transferred (copied) from the user's address space for write operations. For example,

`nread = read(desc, buffer, nbytes)`

reads data into a system buffer and copies it to the user's address space starting at *buffer* up to 512 bytes at a time until *nbytes* are read in. System buffered I/O does not require the locking of the user's data segment during each I/O operation; however, it does require the copying of data from supervisor address space to user address space. It also limits each I/O transfer to 512 bytes at a time, this being the size of a system buffer.

Physical I/O involves the transfer of data directly to/from the particular I/O device from/to the user's address space. Having defined the I/O transfer mode for the file *desc* as physical, the system call:

`nread = read(desc, buffer, nbytes)`

transfers data directly from the physical device to the user program's data space. This data transfer does not make use of system buffers but does require that the user's data segment be locked in memory for the duration of the I/O transfer. Too many large simultaneous physical I/O transfers may have an effect on total system performance for time-sharing processes. If the data transfer specified does not start on a device block boundary, the transfer is broken up into a combination of buffered and

physical I/O transfers automatically. Besides being able to selectively decide which file(s) are to be accessed using physical I/O, one may also decide to use physical I/O transfers for all of a process's open files where possible using:

```
setio(-1, 1)
```

Reverting back to normal buffered I/O for all files is achieved using:

```
setio(-1, 0)
```

Asynchronous I/O involves the transfer of data directly to/from the particular I/O device from/to the user program's address space without requiring the user program to wait for its completion. In fact, a user program may initiate up to four I/O's to/from an arbitrary number of files before having to wait for the completion of any of them. The system call:

```
bufdes = read(desc, buffer, nbytes)
```

initiates the I/O transfer returning *bufdes*, the buffer descriptor to be used in a subsequent check on the status of the I/O transfer. The

```
status = statio(bufdes, statbuf, wflag)
```

system call is used to check on the status of a particular I/O transfer. Here the value of *wflag* determines whether to wait for the completion of the I/O transfer or not. If the I/O transfer is complete, the status of the I/O is returned in the three-word buffer descriptor *statbuf*. All asynchronous I/O transfers must start on a device block boundary but need not necessarily end on a block boundary. The system waits for the completion of all outstanding I/O transfers before terminating a process.

5. Messages

The use of messages for interprocess communication is at the heart of the MERT system. These facilities have been made available to the UNIX user program by means of a number of primitives. At present, two sets of primitives are available. The first set includes a simple send and receive capability. The second set of primitives provides a more powerful facility with some protection and enhanced capabilities. The design of the latter was done to produce a set which was compatible with the USG-UNIX Generic 3 version of messages [8].

The first set of primitives will be discontinued at some time in the future, since its capabilities are now covered by the second set. They are described here for completeness. To send a message to another process, the

```
msgsend(msgbuf, msgsize)
```

primitive can be used. The maximum size message which can be sent is 112 words. The message buffer *msgbuf* contains a six-word header as well as the body of the message. The sender need only fill in the process number of the process to which the message is being sent. The message type is always -3 for user messages. The sender must fill in the *msident* word in the message header so that acknowledge messages may be identified. The sender must specify the size of the message in *msgsize*. To receive a message, two primitives are available:

```
msgrecv(msgbuf, msgsize)
```

```
msgrecn(msgbuf, msgsize)
```

The first one receives a message up to *msgsize* words long into the buffer provided by the user program, *msgbuf*. Only type -3 messages are received. If no messages are on the process input queue, the process roadblocks. The second primitive, *msgrecn* is provided so that the process will not roadblock if no messages are on the process input queue.

The second set of message primitives were designed in collaboration with the builders of USG-UNIX Generic 3 to provide a more complete set of message facilities. A process is given the ability to enable or disable message reception by means of the

```
msgenab()
```

```
msgdisab()
```

primitives. Message reception remains enabled across *exec*, but not across *fork*. To send a message to another process, two primitives are available:

```
send(buf, size, topid, type)
sendw(buf, size, topid, type)
```

The only difference between the two is that the *send* primitive returns immediately if system message buffers are full but the *sendw* primitive will cause suspension of execution until sufficient system buffer space is available. The size of the message in the buffer *buf* is specified by *size* bytes. The destination process is given by *toid* whereas the type of message is given by *type*. To receive a message, two primitives also are available:

```
recv(buf, size, mstructp, type)
recvw(buf, size, mstructp, type)
```

Here return is immediate from *recv* even if no message exists on the input queue. A *type* of 0 indicates that the first message on the input queue is to be retrieved. A *type* from 1 to 128 indicates that the first message of the required type is to be received. The type of message received and the process from which the message was received is returned in the structure pointed to by *mstructp*. In all cases of send and receive, the total length of the message sent/received is returned. Further details of the message facilities may be found in section F of the MERT Programmer's Manual [6].

6. Process Ports

Knowing the identity (process identifier) of a process gives another process the ability to communicate with it. The concept of process ports was introduced in the MERT system to satisfy the requirements of communication between unrelated processes. A process port can be considered to be a globally known "device" to which a process may attach itself to allow other processes to communicate with it without knowing its process identifier. By means of the system call:

```
pid = sysproc(portnum, flag)
```

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, depending on the value of *flag*. Having identified itself globally by connecting to a process port, other processes may communicate with it by sending messages to it through the port. A system call:

```
msgport(msgbuf, msgsize)
```

is provided for this purpose. Once the communication channel is established, these cooperating processes may now also communicate by means of the other interprocess communication facilities, including events and shared memory. Upon the death of a process owning a port, that port is freed up. Once a port is claimed, other processes may not re-allocate that port.

7. Shared Memory

One of the main means of interprocess communication in MERT is the use of shared memory achieved via named segments. Each supervisor-user process consists of a number of segments both in supervisor and user mode address space. A normal UNIX program in user space consists of a minimum of two segments, a combined text and data segment and a stack segment. If text and data are separated, then a total of three segments is used. If I and D space are separated, one I-space segment and two D-space segments are used. A segment is defined as a logical piece of contiguous address space, 32 words to 32K words in size. Since the PDP-11 memory management unit breaks up the 32K word address space into eight pieces of 4K words each, both in I-space and D-space, a user program may have a maximum of eight segments in each address space at one time. Thus if I and D space are not separated, a maximum of six user specified segments may be added to the user program. If I and D space are separated, a maximum of seven user text segments and a maximum of six data segments may be added. The text segments are typically used for public libraries (see later section). The data segments which may be manipulated by the user program are described here.

A user program may add six data segments to his address space at one time in addition to the text, data and stack segments, provided the total data address space does not exceed 32K words. The user

program may manipulate more than six segments but it can only have six in its active address space simultaneously. Six function calls are provided to control access to these segments. They all require a common five-word structure which defines the segment characteristics:

```
struct segstruct {
    int         segname[2];
    char        perm;
    char        breg;
    int         segsize;
    char        *segaddr;
};
```

The segment name *segname* is a 32-bit quantity and must be specified when a segment is being referred to. Segments are normally managed by maintaining a table of the above structures, one for each segment. If the name of a segment is not known, the segment descriptor, *segd* may be used to identify it. This segment descriptor refers to an entry in the process PCB table of user segments, a value of zero being the first entry. Thus if the segment name *segname* passed as an argument to the system call is less than 256, it is taken to be a segment descriptor, *segd*.

The *makeseg* primitive is used to create a new segment. The name of the segment must be zero since the segment does not exist. The user may specify the base register *breg* in which he wishes the segment to start or he may let the system choose the next available base register. The size of the segment *segsz* must be given in bytes. One may determine the access permissions for the segment for both the parent process and for any children which are subsequently spawned. The starting virtual address is returned in *segaddr*. The segment is inherited across *fork*'s as well as across *exec*'s. *Getseg* is used to get an already existing and named segment into the user's address space. The name of this segment may have been passed by an interprocess message. *Rmovseg* is used to delete a segment from the user's address space. *Connseg* is used to put a segment in the process' PCB into the user's address space. A segment may be temporarily disconnected by means of the *discseg* primitive. In the case of segments put in the process' virtual address space before it was invoked, one may obtain the name of the segment by means of the *getsnam* primitive. This is typically used for public library segments.

8. Event Mechanism

Events have been implemented in the MERT system to achieve the communication of a small amount of information between processes. Up to 16 different events may be sent and received by processes. However only 8 of these are available to user programs (bits 0 to 7 of the event word represent the 8 events). These events are typically used to synchronize the execution of co-operating processes sharing a common data segment. Events differ from the UNIX "signals" in that signals are caught asynchronously, i.e. the process execution is interrupted to cause a transfer of program control to the user designated routine to process the signal. In the case of events, no asynchronous activity is apparent to the user program. The primitive

```
sendev(process, event)
```

is used to send the event pattern in *event* to the process *process*. The receiving process uses the primitive

```
event = waitev()
```

to determine which event(s) are received. It roadblocks until one or more events are received. The event mechanism described here is sufficient to deal with situations requiring completely synchronous communication.

9. Call Mechanism

The normal mechanism used in UNIX to bring another process into execution is to spawn a process by making a copy of the parent process using the

```
chid = fork()
```

primitive. This primitive has two returns. The parent process returns with the process ID of the child

process *child*. The child process returns with *child* equal to zero. The child process will then typically overlay itself with the image of the new program by means of the

```
execl(name, arg1, arg2, ..., argn, 0)
```

primitive or the

```
execv(name, argv)
```

primitive. Here *name* is the pathname of the program to be executed and *arg1* to *argn* are the arguments to be passed to the program. For a variable list of arguments, it is usually more convenient to pass a null terminated list of arguments. The parent process typically invokes a

```
child = wait(&status)
```

primitive to wait for the death of the child process. Both the child process identifier *child* and the status of the process *status* are returned.

It is sometimes desirable to be able to execute a new program without making a copy of the parent process since the copy is not used typically except to perform the execute function. It is much more efficient then, to combine the *fork* and *wait* in the parent process and do the equivalent of *exec* in the child process. The *call* primitive spawns a child process without making a copy of the user segments. It sets up the user address space with the named file and then transfers to the beginning of the core image of the file. Files remain open across the *call* primitive. Ignored signals remain ignored across *call*, but signals that are caught are reset to their default values. Any user segments which are created by the parent process remain set up across the *call* primitive.

There are two means of invoking the call mechanism:

```
lcall(name, arg1, arg2, ..., argn, 0)
```

and

```
vcall(name, argv)
```

The status of the child process is returned in *status* by issuing a

```
wait(&status)
```

primitive in the parent process. All other arguments have the same meaning as for the *execl* and *execv* primitives. The *lcall* primitive is useful when a known file with known arguments is being called. The *vcall* version is useful when the number of arguments is unknown in advance.

10. Process Locking

A supervisor or supervisor-user process can be guaranteed response of the order of a few milliseconds only if it is in core at the time which an event arrives for it. Normally user processes are swapped out to secondary storage if not active for some time. To guarantee real-time response to a user process it must be locked in memory using the

```
plock(flag)
```

primitive. This marks all user and supervisor segments for this process non-swap for a non-zero value of *flag*. The process may be unlocked with a zero value of *flag*. To guarantee good response, a process should also run at a high software priority.

11. Semaphores

The semaphores provided in the MERT/UNIX supervisor are identical to those provided by the USG-UNIX Generic 3 system [8]. Semaphores are useful for the synchronization of processes and for the control of access to resources. For historical reasons, two versions of semaphores are provided for these purposes. One is the lock-unlock type; the other is the P-V or counting type. A semaphore must be allocated by a process before it can be used. The semaphore's access permission and type are established by the first process to allocate it and remains in effect until freed by every process that has it allocated. Allocated semaphores are inherited across *fork* but not across *exec*.

Semaphores are allocated and freed by means of

`number = allocsem(number, type)`

and

`freesem(number).`

Here a *negative number* implies that an available semaphore from the system pool is chosen and allocated. When allocating a semaphore, the *type* specifies the function for which it is to be used and what class of processes are eligible to use the semaphore. A value of 0, 1 or 2 allocates a *lock-unlock* semaphore with an access class of anyone, same userid only, and same groupid, respectively. A value of 3, 4 or 5 allocates a P-V semaphore for use by anyone, same userid only, and same groupid, respectively.

Lock-unlock semaphores are manipulated using the following primitives:

`lock(number)`
`unlock(number)`
`tlock(number).`

Lock locks the semaphore specified by *number*. If the semaphore is already locked, the process suspends execution until it is unlocked. *Unlock* unlocks the semaphore and wakes up any process waiting for it to be unlocked. *Tlock* locks the semaphore if it is not already locked but returns immediately if already locked.

P-V semaphores are manipulated using the following primitives:

`p(number)`
`v(number)`
`test(number).`

The *P* operation decrements the semaphore value by one if positive but suspends execution if zero, waiting to become positive. The *V* operation increments the semaphore value by one and wakes up any process waiting for the value to become positive. The conditional *P* operation is achieved by the *test* primitive. Execution of the process is not suspended even though the value was already zero.

The mechanisms described here are rather general in that they are meant to satisfy a large class of users. Thus the amount of code required for implementation is quite large.

12. Public Libraries

User programs often share a number of library routines. If the size of these routines is significant and there are many user programs sharing the same routines, multiple copies will exist in memory. To avoid this, public libraries were implemented in the user program environment. This saves a significant amount of physical memory in cases where a number of programs need to share a large set of routines as in the case of a centralized data base manager. Data as well as text may be shared either in one segment or separate segments. When libraries are formed using the `ldp[6]` command, the starting base registers must be specified. The output is placed in a type '0401' file which uses a header block of 512 bytes to specify the segments which are included. The final user program, also of type '0401' may include up to three public libraries. The segments which make up the 'a.out' file including those of the public libraries must not overlap in virtual address space.

A common problem associated with public libraries is that when it is reformed, i.e. old subroutines modified or new subroutines added, all programs which referenced it had to be link-edited again to make the appropriate connection to the subroutine entry points in the public library. The implementation of public libraries in the MERT system makes use of transfer vectors at the beginning of the public library through which subroutine transfers are affected. Version numbers are associated with public libraries. The version number is only changed if entry points are removed from the library. Only if the version number is changed is the link-edit of all user programs using the library required.

The following sequence of commands demonstrates how a sample program which includes two public libraries may be formed. The total shell script is given by:

```

cc -c -O ltst.c publib1.c publib2.c
sh lib1sh
sh lib2sh
ldp ltst.b

```

The two public libraries are formed by the scripts in *lib1sh*:

```

ld -r -x publib1.o -lc -l
ldp publib1.b

```

and *lib2sh*:

```

ld -r -x publib2.o publib1
ldp publib2.b

```

The final user program is formed using the following specification file *ltst.b*:

```

mode:          ud
ifile:         /lib/crt0.o ltst.o
publib:        {
                /src/mert/publib/publib1
                /src/mert/publib/publib2
            }
ofile:         ltst
stack:         02400

```

The user may now invoke the *ltst* command from the shell level as any user level command. The MERT/UNIX supervisor is able to execute the file *ltst* loading the libraries only if they are not already in memory.

13. Implementation

The MERT/UNIX supervisor is part of each user program which runs in user mode as a "UNIX" program. Each "UNIX" process is really a supervisor/user process. The supervisor code and some data (common buffers and open file pointers) are shared amongst all UNIX-user processes. In supervisor-mode, each process consists of at least the following four segments:

Process Control Block (PCB)	read only
supervisor shared code	read only
supervisor shared data	read/write
stack segment	read/write

The PCB contains all of the state information of the process. The contents of the PCB are described in further detail in section A of the MERT Programmer's Manual [6].

The supervisor code segment contains all of the routines necessary to implement the functionality of the UNIX time-sharing system. The supervisor serves the purpose of catching all traps from the user, including the system calls, and translating these into MERT system requests. It also maintains state tables for outstanding I/O requests and the status of all open files. Almost half of the system calls of the standard UNIX system are concerned with file system requests. Each one of these requires the building of an interprocess message to send to the file manager process. As a simple example, the "chdir" system call results in a message being built with the current directory of the user process, the new owner of the file and the name of the file as data in the message. The file manager process validates the request, gets the inode of the given file and checks the permissions of the requesting process against those of the file to determine if the owner of the file can be changed. The file manager process then sends an acknowledgement message back to the UNIX supervisor with a success/fail indication. In the case of a read file system call, the supervisor first checks to see if the requested data is in an in-core buffer. If it is, no request is sent to the file manager. Other system calls such as the "sbrk" system call can be handled completely in the supervisor by making MERT system service calls to the kernel.

The supervisor data segment is a shared segment among all UNIX processes. It contains all of the file block buffers (512 bytes each) to enable the sharing of files between processes. All open files are

maintained here to keep offset pointers into files and an open file count. If two processes are reading the same file, each will get different parts of the file since the offset pointer into the file is updated with each read. All read-only data used by each process is maintained in the shared data segment as well.

The supervisor stack segment is 1024 bytes in size and corresponds to the UNIX system user block. It contains process specific information such as open files, current directory, and other state information. The upper part of the block is used for a downward growing stack.

Pipes in the MERT/UNIX supervisor are implemented using shared segments. When a pipe is created, a 1024 byte segment is created and put in supervisor address space. The segment has a three-word header to maintain a read pointer into the segment, a byte count of number of bytes in the pipe, a count of the number of outstanding opens and a lock byte to synchronize the readers and writers on the pipe. The remainder of the segment is used as a circular buffer to hold data written on the pipe. When the process which created the pipe forks, the pipe segment count is incremented to indicate that two processes now share this segment. A process may have up to eight pipe segments. However only one pipe segment may be in supervisor address space at one time.

The implementation of most system calls is straightforward. One exception is the handling of signals. Signals may be caught in the user program and in this respect the signal system call is handled by setting a vector in the user block segment. The sending of signals is handled by sending an interprocess message to the receiving process with the signal in the body of the message. The event handler receives the message asynchronously and the signal handling routine processes the signal in the same manner as in the standard UNIX time-sharing system.

The "exec" system call in the MERT/UNIX supervisor has some additional capability over that provided by the standard UNIX time-sharing system. It recognizes type '0401' files as being executable files with public libraries included. Up to three public libraries may be included in this file. The text, data and stack segments are loaded as normally done and the library segments are loaded as needed if they are not already in memory. Each library has a version number associated with it to guarantee compatibility between the main user program and library routines. The nature of the MERT file systems permits the segments to be read with one I/O operation each.

The user program itself consists of a number of segments. The user has a great deal of flexibility as to how to manage his virtual address space. Typical UNIX user programs have one segment for code and data (type '0407') and one segment for the stack. The code and data start at the lowest address and expand upward. The stack starts at the highest address and expands downward. For a large program, shared by many users, the text segment may be shared (type '0410'). This gives the user program three segments. If the program becomes very large, then the user may wish to use separated I and D space (type '0411') so that 128K bytes of address space are available to the user. For greater flexibility, the user must use the *ldp* [6] command to create a type '0401' process image in a file. The options provided with the *ldp* command are quite extensive. For instance, the user may combine the code, data and stack portions of the program into one segment and also specify the starting virtual address of this segment (it need not be 0).

14. Uses

The following are some examples of the uses of the MERT system features by user level programs.

14.1 Testing New Processes

The MERT system offers a convenient environment for testing out new kernel-mode as well as supervisor-mode processes. To start up a new process from the UNIX system command level, one may invoke:

```
run [-b] [-f] pfile
```

The optional flag *-b* determines whether the process whose image is in the file *pfile* runs asynchronously or not, i.e. whether the parent process waits for the death of its child process or not. The *-f* flag determines whether the process manager sends a message to the child process to get it started or not.

To test out a new character device driver, one must form the process image in *lprcdx* where *x* corresponds to the major device number. To load the process into memory, one need only invoke the

```
run -b /prc/cdx
```

command. One may now *login* on the teletype device corresponding to the process just loaded. A core dump will be produced in */cdmplecdx* if a breakpoint trap is executed in the process. This may now be debugged using the modified *adb* debugger.

Another frequent use of the process testing feature of the MERT system is for debugging a new version of the UNIX/MERT supervisor. This is required to test out new or modified system calls. A convenient test can be performed using the

```
run unix
```

command, where the file *unix* contains an image of the new UNIX supervisor process. This results in another *login* process being started up. After logging in to the new supervisor, one may test out the new system calls. When testing is complete, hitting the CNTRL D key on the terminal will exit the user back to the previous UNIX supervisor. A new UNIX supervisor may be installed by means of the following command sequence:

```
mv /prc/unix /prc/ounix;mv unix /prc/unix;sync
```

The old version of the UNIX supervisor is saved in case of problems with the new supervisor. Note that the old and new supervisors do not share buffers or file descriptors. This may result in some unexpected side effects.

14.2 File System Primitives

The file system primitives added to the MERT/UNIX supervisor are directly accessible from the UNIX shell command level by invoking two primitives with the same names. Thus

```
falloc fname nblocks
```

allocates *nblocks* 512 byte blocks to the file *fname*. The resulting file is contiguous with an initial size of 0 bytes. The blocks are pre-allocated and the file may not be expanded.

The

```
fmove - fname [... fnamen]
```

command moves the named file[s] into contiguous secondary storage areas. The files are not marked contiguous and thus may grow dynamically. This is a useful command to guarantee the allocation of consecutive blocks to a file for large data transfers using physical or asynchronous I/O with minimum latency and transfer time. The '-' argument is passed along as a flag to the *fmove* system call to ensure that the file is moved even if contiguous space does not exist below (lower block number) where the file currently resides.

14.3 Physical I/O

Two general purpose commands have been provided at the UNIX shell level to make use of physical I/O transfers. One may precede a command with a physical I/O command, much the same as one would do with the *time* command to time a command. Thus:

```
pio cc file.c
```

compiles the file *file.c* using physical I/O wherever possible rather than buffering all data in the system first. This typically reduces the system time for a command and thus the overall execution time. A second command:

```
pcp file1 file2
```

can be used to copy *file1* to *file2* with the same effect as the *cp* command. However, *pcp* uses physical I/O with a 5120 byte internal buffer. This is an efficient means of copying a complete file system from one logical file system to another.

14.4 Asynchronous I/O

Early versions of the MERT/UNIX supervisor did not provide read-ahead for files. This did not appreciably affect programs using disk I/O. However the lack of read-ahead was evident in the 'tp'

program, especially when dealing with DECTape. Thus asynchronous I/O was added to the 'tp' command at the user level to achieve the effect of read-ahead.

Another frequent use of asynchronous I/O is to provide double buffering at the user program level when there is a requirement to collect data at a rate where delays between synchronous reads would result in the loss of data. In these applications, the typical procedure is to use two buffers for reading and one for writing to a contiguous file. Thus there are always two outstanding reads. As soon as one read completes, an asynchronous write is initiated and another asynchronous read is initiated. Here the rate of data collection is limited only by the disk latency and transfer rate. The larger the buffers used, the less important the disk latency is.

14.5 Error Logging

When I/O errors occur in the system, it is helpful to be able to record the time and date of these errors as well as the contents of the device registers. This aids in tracking down hardware faults and hence minimizing the time to repair or even avoiding fatal I/O errors by repairing the device beforehand. The errors caused by all device drivers in the MERT system are recorded by a user program which connects itself to a known port (port number 0) and collects all error messages from the device driver processes. Each message contains the contents of all of the device registers as well as the number of retries necessary for a successful I/O operation. All relevant information concerning the error is written into a file *leiclerrlog* in ascii format. This file may be examined periodically for the occurrence of new errors.

14.6 Scratch Pad Data Transfer

The use of pipes in the MERT system involves the copy of data from user to supervisor address space in the writer program and from supervisor address space to user address space in the reader program. It is much more efficient to use a common data segment between the two communicating processes. This technique has been used by the DIR/ECT project [9] to achieve efficient interprocess data transfers. The interprocess event mechanism is used to provide the required synchronization between the writing and reading processes. The writing process writes data directly into the shared segment starting at an address maintained in a 20 byte header at the beginning of the shared segment. A read pointer is also maintained to point to the data which may be read by the reading process. Upon reading, the reader is passed an address pointer to the start of data. Thus there is actually no copying of data involved. Timing measurements indicate that this technique requires a factor of 4 or 5 less running time than by using pipes.

14.7 The Extended Shell

The basic shell as written by S. R. Bourne has been modified by N. J. Kolettis [10] to support an expanded set of builtin functions consisting of the *indigenous* set and a special set of *implanted* ones. Builtin functions were added to the shell to provide better execution efficiency of certain shell procedures by eliminating the forking and executing of processes. The special set includes a shell interface to the interprocess communication facilities of the MERT system, e.g. shared segments, ports, messages, events and semaphores. This makes all of the unique features of the MERT system available at the user's fingertips. The resulting shell, called the *Extended Shell*, is a real-time interpreter that can be used to run shell procedures in real-time environments performing essentially as fast as C programs. This shell has been used in a Bell System project which required fast response for its users. It gives the user real-time capabilities in a language (extended shell language) that is easy to learn and use. As an example,

```
conport $portno
```

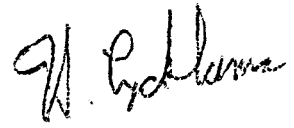
connects the current process to the port specified by the number *portno*. Another process may then send a message to the process connected to this port.

Acknowledgements

The MERT/UNIX supervisor was written during the initial design and implementation of the MERT system in collaboration with D. L. Bayer. The current version of the MERT/UNIX supervisor has benefited substantially from the efforts of T. M. Raleigh who has updated the supervisor by making it more compatible with the current versions of the supported UNIX systems, making it more portable as

well.

HO-3114-HL-hl

A handwritten signature in black ink, appearing to read 'H. Lycklama', written in a cursive style.

H. Lycklama

References

- [1] Lycklama, H. and Bayer, D. L., A Structured Operating System for a PDP-11/45, TM-75-1352-4.
- [2] Bayer, D. L. and Lycklama, H., MERT - A Multi-Environment Real-Time Operating System, TM-75-1352-7.
- [3] Lycklama, H. and Bayer, D. L., The MERT Operating System, TM-78-3114-3.
- [4] Thompson, K and Ritchie, D. M., The UNIX Time-Sharing System, Comm. ACM 17, (July 1974), p365.
- [5] Thompson, K. and Ritchie, D. M., The UNIX Programmer's Manual, May 1976.
- [6] Lycklama, H. and Bayer, D. L., The MERT Programmer's Manual, October 1977.
- [7] Lycklama, H., File System Structures for Real-Time Applications, TM-78-3114-5.
- [8] Brandt, R. B., Implementation of Semaphores and Messages in UNIX, MF-76-8234-076.
- [9] Laur, J. E., Scratch Pad Data Transfer in the MERT System, private communication.
- [10] Kolettis, N. J., Extended Shell - A Potential Real Time Interpreter, TM-77-4145-01.

Appendix A

```
/*  
 * This table is the switch used to transfer  
 * to the appropriate routine for processing a system call.  
 * Each row contains the number of arguments expected  
 * and a pointer to the routine.  
 */
```

```
int    sysent[]  
{  
    0, &nullsys,          /* 0 = indir */  
    0, &rexit,            /* 1 = exit */  
    0, &fork,             /* 2 = fork */  
    2, &read,             /* 3 = read */  
    2, &write,            /* 4 = write */  
    2, &open,             /* 5 = open */  
    0, &close,            /* 6 = close */  
    0, &wait,             /* 7 = wait */  
    2, &creat,            /* 8 = creat */  
    2, &link,             /* 9 = link */  
    1, &unlink,           /* 10 = unlink */  
    2, &exec,             /* 11 = exec */  
    1, &chdir,            /* 12 = chdir */  
    0, &gtime,            /* 13 = time */  
    3, &mknod,            /* 14 = mknod */  
    2, &chmod,            /* 15 = chmod */  
    2, &chown,            /* 16 = chown */  
    1, &sbreak,           /* 17 = break */  
    2, &stat,             /* 18 = stat */  
    2, &seek,             /* 19 = seek */  
    0, &getpid,           /* 20 = get process ID */  
    3, &smount,           /* 21 = mount */  
    1, &sumount,          /* 22 = umount */  
    0, &setuid,            /* 23 = setuid */  
    0, &getuid,           /* 24 = getuid */  
    0, &stime,            /* 25 = stime */  
    3, &ptrace,           /* 26 = ptrace */  
    0, &alarm,            /* 27 = alarm */  
    1, &fstat,            /* 28 = fstat */  
    0, &pause,            /* 29 = pause */  
    1, &nullsys,          /* 30 = smdate; inoperative */  
    1, &stty,             /* 31 = stty */  
    1, &gtty,              /* 32 = gtty */  
    2, &saccess,           /* 33 = access */  
    0, &nice,              /* 34 = nice */  
    0, &nullsys,          /* 35 = sleep; inoperative */  
    0, &sync,             /* 36 = sync */  
    1, &kill,             /* 37 = kill */  
    0, &getswit,          /* 38 = switch */  
    0, &nosys,            /* 39 = x */  
    0, &tell,              /* 40 = tell */  
    0, &dup,               /* 41 = dup */  
    0, &pipe,             /* 42 = pipe */  
    1, &times,            /* 43 = times */  
    4, &profil,           /* 44 = prof */  
}
```



```
0, &nosys, /* 45 = tiu */
0, &setgid, /* 46 = setgid */
0, &getgid, /* 47 = getgid */
2, &ssig, /* 48 = sig */
4, &messag, /* 49 = message */
0, &nosys, /* 50 no system call */
0, &nosys, /* 51 no system call */
0, &nosys, /* 52 no system call */
0, &nosys, /* 53 no system call */
0, &nosys, /* 54 no system call */
0, &nosys, /* 55 no system call */
0, &nosys, /* 56 no system call */
1, &pwbsys, /* 57 pwbsys - udata */
2, &scall, /* 58 = call,vcall */
0, &nosys, /* xx no system call */
0, &nosys, /* xx no system call */
0, &nosys, /* xx no system call */
2, &lflags, /* 62 = locks */
0, &nosys, /* 63 = no system call */
0, &nosys, /* 64 no system call */
2, &pcreat, /* 65 = create new process */
2, &msgrecv, /* 66 = receive message */
1, &msgsend, /* 67 = send message */
1, &sproc, /* 68 = system process */
1, &msgport, /* 69 = send message thru a port */
2, &scall, /* 70 = call routine */
1, &userseg, /* 71 = user segment */
0, &nosys, /* 72 */
1, &sendevent, /* 73 = send event(s) to user */
0, &waitevent, /* 74 = wait for event for user */
1, &setio, /* 75 = set I/O mode */
2, &statio, /* 76 = get I/O status */
4, &flalloc, /* 77 = allocate contiguous space for a file */
3, &fmove, /* 78 = lock and move file */
0, &proclock, /* 79 = process lock */
0, &qwait, /* 80 wait and return if no child */
1, &qsleep, /* 81 qsleep system call */
0, &nosys, /* 82 no system call */
0, &nosys, /* 83 no system call */
0, &nosys, /* 84 no system call */
};
/*
 * Dummy entry for illegal system calls
 */

int badent[] {
    0, &nosys
};
```