1070

MERT PROGRAMMER'S MANUAL

First Edition

H. Lycklama

D. L. Bayer

October, 1975

INTRODUCTION TO THIS MANUAL

This manual provides a description of the internal features of the MERT operating system. It is meant to be used as a supplement to the UNIX PROGRAMMER'S MANUAL. A more general overview of the MERT operating system is provided by the two technical memoranda:

A Structured Operating System for a PDP-11/45 - TM-75-1352-4.
MERT - A Multi-Environment Real-Time Operating System - TM-75-1352-7.

Within the area it surveys, this manual attempts to be as complete and timely as possible. A conscious decision was made to describe each program in exactly the state it was in at the time its manual section was prepared. In particular, the desire to describe something as it should be, not as it is, was resisted. Inevitably, this means that many sections will soon be out of date.

This manual is divided into six sections:

A. Supervisor Process Calls
B. Kernel Process Calls
C. Inter-Process Message Formats
D. File System (and Utilities)
E. MERT-UNIX Programs
F. MERT-UNIX System Calls

The PCB (Process Control Block) of a supervisor-user process is described in section A. A supervisor-user process has entries into the kernel by means of EMT traps. Each one of these is described in detail. The binary object code for each routine is kept in the library file "/lib/libe.a".

The kernel process header is described in section B. A kernel process has entries into the kernel by means of another set of EMT traps. Each of these is described in detail here. The binary object code for each routine is kept in the library file "/lib/libk.a".

Inter-process communication is achieved mainly by means of messages. The header of a message is described in section C along with the contents of the various message types which are recognized by the basic system processes. These processes include kernel I/O drivers, file manager, process manager, memory manager and system process scheduler.

The format of a file system is described in section D along with the layout of each file i-node. The utility programs which deal with the file system are also described here. For those which are not described, the reader is referred to the UNIX PROGRAMMER'S MANUAL.

Section E describes the various utility programs which are used to build special MERT files such as process images and the boot image. It also includes desriptions of programs which will run

INTRODUCTION TO SUPERVISOR EMT TRAPS

The interface between a supervisor process and the kernel consists of a read only data segment called the process control block (PCB), approximately fifty EMT traps, and a few messages associated with creation and termination of a process and its children.

The PCB describes the process virtual address space (both supervisor and user modes), defines entry points for handling interrupts (events) and faults, and provides the scheduler with space for saving the state of the machine when switching between processes. The structure of the PCB is:

```
struct pcb {
    int     p_pnum;         /*process number*/
    int     p_parent;       /*Process number of parent process */
    char    p_prior;        /*Initial scheduling priority*/
    char    p_chan;         /*Process control channel */
    char    p_name[8];      /* ASCII name of process */
    int     p_ttg;          /*Scheduler saves Time To Go of process time slice on premption */
    int     p_slice;        /*Process time slice in 1/60 seconds*/
    int     p_size;         /*Total number of bytes in segment table*/
    struct {
        int     ktime;      /*Total time spent in kernel mode while process was active*/
        int     stime;      /*Total time spent in supervisor mode while process was active*/
        int     utime;      /*Total time spent in user mode while process was active*/
    } p_times;
    char    p_timeout;      /*Inhibit context change flag for the scheduler*/
    char    p_wait;         /*Hold in memory until time slice runs out flag to the scheduler*/
    char    p_tflag;        /* If non-zero a terminate message of type p_ttype will be sent to parent upon the death of this process*/
    char    p_ttype;        /*Message type to send to parent on death of this process*/
    char    *p_tident;      /* Message Ident to be sent to parent on death of process*/
    int     p_save;         /* Scheduler saves pointer to preempted process here*/
    int     p_semafor;      /* not used */
    int     p_dummy;        /* not used */
```

under MERT-UNIX making use of the new "system call's" added to the MERT version of the UNIX supervisor.

Section F describes all of the new "UNIX system call's" added to the MERT-UNIX supervisor. The binary object code for each routine is kept in the library file "/lib/libr.a".

Most sections begin with an introduction section. Each section consists of a number of independent entries of a page or so each. The name of the entry is in the upper corners of its pages, its preparation date in the upper middle. Entries within each section are alphabetized (except for section C). The page numbers of each entry start at 1.

All entries are based on a common format, not all of whose subsections will always appear.

The name section repeats the entry name and gives a very short description of its purpose.

The synopsis summarizes the use of the program being described. A few conventions are used, particularly in the Commands section:

Boldface words are considered literals, and are typed just as they appear.

Square brackets ([ ]) around an argument indicate that the argument is optional. When an argument is given as ``name'', it always refers to a file name.

Ellipses ``.|.|.'' are used to show that the previous argument-prototype may be repeated.

A final convention is used by the commands themselves. An argument beginning with a minus sign ``-'' is often taken to mean some sort of flag argument even if it appears in a position where a file name could appear. Therefore, it is unwise to have files whose names begin with ``-''.

The description section discusses in detail the subject at hand.

The files section gives the names of files which are built into the program.

A see also section gives pointers to related information.

A diagnostics section discusses the diagnostic indications which may be produced. Messages which are intended to be self-explanatory are not listed.

The bugs section gives known bugs and sometimes deficiencies. Occasionally also the suggested fix is described.

```
        int     *r_leng;    /* Segment length in 32 word blocks
                             */
        char    r_ucnt;     /* Number of users */
        char    r_status;   /* Segment status flags */
        char    *r_disk;    /* Starting block on the swap dev-
                               ice */
        int     r_name[2]   /* Unique 32 bit name */
};
```

The sharing of segments by independent cooperating processes is accomplished via the r_name entry in the RSDE table. The processes need only create a segment with the same unique name in order to share the same physical segment. The system convention for establishing unique names for segments is based on the premise that shared segments will contain some initial data, and this data will reside in a file. The name is simply the absolute disk address (major, minor device and block number) of the first block of the initialized data. For processes which have a parent child relationship, sharing is accomplished by passing a segment ID between parent and child.

The bits of pstat are defined as follows:

bit 15	1 if segment is active. Symbolic name pcbnn for need now.
bit 14	1 if segment is needed next time the process is activated. Symbolic name pcbnxt for next.
bit 13:12	Address space: 1 for supervisor, 2 for user
bit 11	1 if D-space
bit 10:8	Starting segmentation register number.
bit 7	1 if segment has been made non-swap by the process
bit 6	1 if memory manager did not load this segment (even though pcbnn or pcnxt were set) because of insufficient swap space. Symbolic name pcbnold.
bit 5	1 if segment is sharable. Symbolic name pcbshare.
bit 4	1 if segment is writeable by the process. Symbolic name pcbwrite.
bit 3	1 if segment is a stack segment. Symbolic name pcbed.
bit 2:0	Access setting for the segment.

In this section of the manual, the statement "segment indexed by segnum" means that the kernel will access the PCB segment table:

```
struct pcb p;
 .....
p.p_tab[ segnum ].pstat ...
```

The kernel procedure on all traps from supervisor and user mode, except break point traps from supervisor (a core dump is produced in /cdmp/p_name), is:

1) The psd saved on the stack is put into the PCB at p_fpsd.
2) For floating point or segmentation traps, the appropriate registers are saved in p_fsav.
3) The fault code is saved in p_fcode:

```
        struct psdd p_topsd; /*Scheduler save pc and ps of
                             process on time out or preemp-
                             tion*/
        struct {
            int    p_toreg[6];
            int    p_ssp;      /*supervisor stack pointer*/
            int    p_usp;      /*User stack pointer*/
            int    p_kssr3;    /*Current state of supervisor
                               and user D space register*/
            double p_fprg[6]; /*Floating point registers */
            int    p_fps;
        } p_tosave;            /*Scheduler saves the context
                               of a process here*/
        int    p_event;        /*Event flags are stored here
                               */
        struct psdd p_evect;   /* ps and pc for vectoring to
                               event handling routine*/
        struct psdd p_evpsd;   /* Interrupted ps, pc are saved
                               here */
        int     p_evmask;      /* Event mask word; a 1 in
                               each bit position enables the
                               corresponding event */
        struct psdd p_fvect;   /* ps, pc for vectoring to su-
                               pervisor fault handling
                               routine*/
        int    p_fcode;        /* Type of fault */
        struct psdd p_fpsd;    /* ps, pc save area */
        int    p_fsav[3];      /* Save area for fault depen-
                               dent parameters - for segmenta-
                               tion faults ssr0, ssr1, and
                               ssr2; for floating point faults
                               FEC, FEA, and floating point
                               status register */
        struct psdd p_emtpsd; /* ps, pc on EMT traps to the
                               kernel */
        int    p_emtsave[6];   /* r0 -r5 are saved here on EMT
                               traps to the kernel */
        struct {
            int    pstat;      /* Segment status word*/
            int    psegid;     /* Segment ID */
        } p_tab[];
    }
```

The segment table p_tab[] defines the supervisor and user virtual address spaces. The table contains a minimum of three entries which, in the order they appear, are: the PCB segment, a supervisor stack segment, and a code segment. The maximum number of entries is currently limited to 48 (no more than 32 can be active at one time). Each table entry defines how a segment is to be accessed by the process (see pstat below). The segments specified by psegid are contiguous pieces of memory and (swap space on disk) which vary from 32 to 32K words in 32 word increments. The segment ID is a pointer to a table in the kernel (the RSDE table) which has the structure:

```
    struct rsde {
        char   *r_ptr;    /* Pointer to memory management
                           tables*/
```

NAME

addseg - add a segment to the process address space

SYNOPSIS

```
(addseg = 4)
addseg(segnum)
```

DESCRIPTION

Addseg searches the process segment table and deactivates (and removes non-swap status from) any other segment which starts at the same virtual address as the entry indexed by segnum. If the new segment is in memory or if space for an uninitialized segment is available, the appropriate segmentation registers (as specified by the flags word of the segment table entry) are loaded. If the new segment is not in memory, the "need next" bit is set in the process segment table, the process status is set to "swap permitted, not ready to run", and control is transfered to the scheduler. When rescheduled non-swap status is re-instated .

ALSO SEE

rmovseg(a)

DIAGNOSTICS

C returns -1 if the segnum points to a null segment table entry. Fault return with code 10 if segnum exceeds the available number of segment table entries.

0 bus error
1 illegal instruction
2 trace trap
3 IOT trap
4 power fault
5 EMT (from user only)
6 trap trap
8 floating point exception
9 segmentation fault
10 invalid emt (from supervisor mode only)

4) The new ps and pc are extracted from p fvect and put on the kernel stack.
5) The c-bit in the new ps is set if the trap is a segmentation fault from supervisor mode.
6) If f code = 10, p fvect is cleared.
7) The kernel executes an rti to give control to the supervisor trap handling routine.

The kernel procedure on the arrival of an event is:

1) The appropriate bit in p event is set.
2) If the corresponding bit in p evmask is set and the psd in p event, is defined, the interrupted ps and pc are posted in p evpsd and an rti is executed to the psd in p evpsd.

The description of all the EMT calling sequences are given for C. In most cases no reference to the assembly language calling sequence is given. One can translate the C call to the assembly call by putting the address of the argument list in r0 and substituting c-bit set for -1 returns. In cases where the assembly language returns are more complex than simple success or fail, a complete description is included.

The message formats for creating and terminating processes are discussed in section C of the manual.

NAME

adduser - increment user count on a process

SYNOPSIS

```
(adduser = 43.)
adduser(process)
int process;    /* process number */
```

DESCRIPTION

Adduser increments the user count on the process specified by process. A 1 is returned from C for a successful call.

SEE ALSO

DIAGNOSTICS

A - 1 is returned from C if the process does not exist.

alockseg - lock a segment in memory and set write back

SYNOPSIS

```
(alockseg = 13.)
alockseg(segnum)
```

DESCRIPTION

The segment indexed by segnum is locked in memory and the segment write back bit is set. This system function should be used for reading into a segment; lockseg should be used for writing out of a segment. See lockseg for unlocking responsibility.

ALSO SEE

lockseg(a), lockid(a), ulockseg(a), ulockid(a)

DIAGNOSTICS

C returns -1 if:

1) segnum points to a null segment table entry
2) segnum points to an inactive segment

Fault return with code 10, if segnum points beyond the end of the segment table.

NAME

alocseg - create a segment

SYNOPSIS

```
(alocseg = 0)
alocseg(segnum,size,partition,0)
alocseg(segnum,size,partition,name)
int     segnum;
int     size;
int     partition;
int     name[2];
```

DESCRIPTION

If name is zero, a segment id and size (words) of swap space are allocated. If name is non-zero and matchs an existing segment name, the segment user count is incremented. If name is non-zero but does not match an existing name, an ID is allocated and set in the blocked state to permit initialization. The ID is returned in the process segment table indexed by segnum. The new segment will not be added to the process address space and thus is not brought into memory. When read into memory, the segment will be loaded into the memory partition specified by partition.

From assembly language, r0 points to a four word block containing segnum, size, partition, and name (the address of a double word name). If the request is successful, the c-bit is clear and r0 contains the new segment id. If the request is partially successful, the c-bit is set and r0 contains the status of the request:

1) r0 = -1 Swap space is saturated
2) r0 = 0 Segment is blocked and the process has been put to sleep on the segment id (named segments only).
3) r0 = ID Segment has been created and must be initialized (named segments only).

A typical sequence to create a named segment is:

1) allocate a named segment id
2) setmap to define virtual address
3) addseg to bring the segment into the process address space
4) read data into the segment
5) unblock the segment

ALSO SEE

addseg(a), setmap(a), sleep(a), unblkseg(a)

DIAGNOSTICS

C returns -1 if system is out of swap space or segment ID's. A return of zero if a named segment has been created in the blocked state (eg is being initialized by another process). Fault return with fault code 10 if segment table entry is non-zero or segnum is greater than 47.

NAME

attach - attach process to interrupt vector

SYNOPSIS

```
(attach = 35.)
attach(process, vector, entry)
```

DESCRIPTION

Attach attaches the process process to the interrupt vector at address vector. The PC, PS pair is loaded with the entry point entry and the priority of the process, respectively. The address of the "jsr" attached to the interrupt is returned in C. This system function is normally only invoked by the process manager to load and enable a new device driver.

SEE ALSO

detach(a).

DIAGNOSTICS

An error is returned if the process does not exist, if accessing the device register gives a bus error or if the interrupt vector is invalid (-1 in C).

NAME

copyseg - make a copy of a segment

SYNOPSIS

```
(copyseg = 21.)
copyseg(segnum, &newid, msident)
int segnum;     /* index into PCB segment table */
int *newid;     /* address of new segment ID */
int msident;    /* message identifier word */
```

DESCRIPTION

Copyseg makes a copy of the segment specified by segnum in memory if possible. If the in-core copy has been successful, a value of 1 is returned in C. If the in-core copy was not successful, but a copy had to be created in the swap area, a value of 0 is returned in C. In this case a message has been sent to the memory manager to make a copy. An acknowledgement message will be returned to this process upon completion of the copy with a message identifier word msident. In both cases the segment id of the new segment is returned in newid. This system function is typically used to make a copy of an existing process.

SEE ALSO

DIAGNOSTICS

If the segment to be copied does not exist or if space could not be allocated for the new segment, a -1 is returned from C.

NAME

clrevent - clear event flag(s)

SYNOPSIS

```
(clrevent = 33.)
clrevent(eflag)
```

DESCRIPTION

Clrevent clears the event flag(s) specified by the bits set in eflag, in the PCB (See event(a) for definition of predefined flag bits). The bits set in eflag are reset in the p_event word in the PCB. The bit(s) to be cleared normally correspond to the event just received.

SEE ALSO

event(a), enevent(a)

DIAGNOSTICS

NAME

cwait - conditional wait for event

SYNOPSIS

```
(cwait = 25.)
cwait(&flag)
int flag;
```

or

```
cwait(0)
```

DESCRIPTION

Cwait causes the current process to give up control (enter the road blocked state) if the value of flag is non-zero; an immediate return occurs if flag is zero. A cwait call with an argument of zero causes the p_cwait location in the PCB to be used in place of flag. P_cwait is set to one by many of the kernel EMT traps: (sleep, sendmsg, sendmsgfrom, sendcpmsg, ioqueuem, getmsg, gettype, event, crdblk, and cwait), and cleared by the kernel EMT traps enevent and clrevent, as well as the occurance of any event. If flag (or p_cwait) is non-zero the location p_wait (also in the PCB) will be set. This will cause the scheduler to keep the process in memory for the remainder of it's time slice.

Cwait should only be used if the process expects the condition causing the process to road block will be cleared up within 200 milliseconds. If a longer wait is expected use crdblk.

Since event interrupts are inhibited while the kernel checks flag, potential timing problems between the "base line" and asynchronous event handler parts of a supervisor process can be resolved. The type of timing problem is illustrated by the buffered I/O in the UNIX supervisor: The "base line" code will set flag to one and initiate a buffer write then call cwait(&flag) waiting for the I/O to complete. If the I/O manages to complete before the "base line" completes execution of the cwait (preemption could occur), the event handler will mark the buffer I/O as done and clear flag. Base line will then complete the cwait call. The kernel will detect a zero flag and return from the cwait preventing the supervisor from road blocking for an event which has already occurred.

SEE ALSO

crdblk(a).

DIAGNOSTICS

NAME

crdblk - conditional road block for event

SYNOPSIS

```
(crdblk = 24.)
crdblk(&flag)
int flag;
```

or

```
crdblk(0)
```

DESCRIPTION

Crdblk causes the current process to give up control (enter the road blocked state) if the value of flag is non-zero; an immediate return occurs if flag is zero. A crdblk call with an argument of zero causes the p_cwait location in the PCB to be used in place of flag. P_cwait is set to one by many of the kernel EMT traps: (sleep, sendmsg, sendmsgfrom, sendcpmsg, ioqueuem, getmsg, gettype, event, crdblk, and cwait), and cleared by the kernel EMT traps enevent and clrevent, as well as the occurance of any event. Since event interrupts are inhibited while the kernel checks flag, potential timing problems between the "base line" and asynchronous event handler parts of a supervisor process can be resolved.

SEE ALSO

cwait(a).

DIAGNOSTICS

NAME

dropseg - drop a segment from a process virtual address space

SYNOPSIS

```
(dropseg = 6)
dropseg(segnum)
```

DESCRIPTION

The segment indexed by segnum is removed from the process virtual address space (the "active" bit is cleared in the segment table status word). The "altered" bit is unconditionally cleared and the "new segment" bit is set. The result of all this bit setting and clearing is that the contents of the segment are totally unpredictable. If the current contents of the segment are no longer required, swapping will be reduced when the segment is reactivated. This system function can be used by a process which dynamically creates subtasks and must assign working storage to the subtask. The contents of the segment can be disgarded when the subtask finishes, but the segment is retained for future tasks.

ALSO SEE

rmovseg(a)

DIAGNOSTICS

C returns -1 if the segnum points to a null segment table entry. Fault return with code 10. if segnum exceeds the available number of segment table entries.

NAME

detach - detach process from interrupt vector

SYNOPSIS

```
(detach = 36.)
detach(process, vector)
```

DESCRIPTION

Detach detaches the process process from the interrupt vector at address vector. The entry point for the process is cleared and the device control register is also cleared. This system function is normally only invoked by the process manager in the process of taking down a device driver.

SEE ALSO

attach(a).

DIAGNOSTICS

An error is returned if the process does not exist or if the interrupt vector is invalid (-1 in C).

NAME

event - send event to a process

SYNOPSIS

```
(event = 32.)
event(process, eflag)
```

DESCRIPTION

Event sends the event(s) specified by the eflag word to the process specified by process. Sixteen flag bits are available, eight of which have the following pre-defined meanings:

```
0100000 wakeup
0040000 timeout
0020000 message
0010000 hangup
0004000 interrupt
0002000 quit
0001000 abort
0000400 init
```

The other eight are user definable. Sending an event causes the system to set the appropriate bit in the process' control table and trigger the programmed interrupt at the processor priority of the receiving process.

SEE ALSO

clrevent(a), enevent(a)

DIAGNOSTICS

If the process specified does not exist, an error is returned (-1 in C).

NAME

enevent - enable event flag(s)

SYNOPSIS

```
(enevent = 34.)
enevent(eflag)
```

DESCRIPTION

Enevent sets the event flag(s) specified by the bits set in eflag, in the PCB p_evmask word (See event(a) for definition of pre-defined flag bits). The bit(s) to be set correspond to the event(s) which the process is enabled to receive. The p_evmask word is cleared upon the receipt of an event to prevent multiple events from being received.

SEE ALSO

event(a), clrevent(a)

DIAGNOSTICS

NAME

freeseg - remove a segment ID from the process segment table

SYNOPSIS

```
(freeseg = 3)
freeseg(segnum);
int segnum;
```

DESCRIPTION

The segment table entry indexed by segnum is zeroed. The segment user count is decremented and if zero, the segment ID is set in the unallocated state and the associated swap space is returned to the system.

ALSO SEE

dropseg(a), rmovseg(a)

DIAGNOSTICS

C returns -1 if the segment table entry indexed by segnum is zero. Fault return with code 10 if segnum exceeds the segment table.

NAME

execute - execute new process

SYNOPSIS

```
(execute = 44.)
execute(pcbbase, dspace, &psd)
int pcbbase;     /* base register for PCB */
int dspace;      /* process d-space bits */
int *psd;        /* pointer to entry points */
```

DESCRIPTION

Execute frees up all currently active segments except for the PCB and sets all the remaining segments as active. Vacated slots in the PCB are squeezed out. Pcbbase specifies which base register is used to point to the PCB (0-7 is supervisor i-space and 8-15 is supervisor d-space). The two low order bits of dspace indicate if d-space is to be turned on for the supervisor (bit 1) and for the user (bit 0). Psd points to three sets of entry point pairs of PS and PC:

event entry point
normal entry point
fault entry point

An INIT event is sent to this process to start it up. No return is possible from this system call. The execute function is used by the NUB process to start up a new supervisor-user process.

SEE ALSO

setdspac(a), event(a).

DIAGNOSTICS

NAME

getime - get time

SYNOPSIS

```
(getime = 39.)
getime(time)
int time[2];
```

DESCRIPTION

Getime returns the time in the kernel into the array time.

SEE ALSO

setime(a).

DIAGNOSTICS

NAME

getchar - get characters from kernel process

SYNOPSIS

```
(getchar = 192.)
getchar(pointer, segid, offset, count, &flag)
int *pointer;   /* pointer to process, channel pair */
int segid;      /* ID of segment being read into */
int offset;     /* byte offset into segment */
int count;      /* byte count */
int *flag;      /* flag word */
```

DESCRIPTION

Getchar returns count characters to the caller in the segment specified by segid starting at offset bytes into the segment. If the segment is a stack segment the offset is from the end of the segment. Pointer points to the pair of words specifying the process number of the kernel process and the logical channel number of the device controlled by the kernel process. The call to the kernel within the library routine returns the byte count. If no bytes are read, the library routine roadblocks the user, waiting for input. If a -1 is returned, an EOT signal was received on input, and control is passed directly back to the user with a zero byte count. In the roadblock state, if a quit or interrupt signal is received, the user is unroadblocked, and the non-zero value of the flag causes an immediate return to the user.

SEE ALSO

putchar(a), setty(a), getty(a)

DIAGNOSTICS

If an error condition is detected (illegal buffer address or size) the negative value of the error code is returned in C.

NAME

getty - get state of tty driver process

SYNOPSIS

```
(getty = 195.)
getty(pointer, buffer)
int *pointer;   /* pointer to process, channel pair */
int buffer[3];  /* 3 word buffer */
```

DESCRIPTION

Getty returns the state of the tty kernel driver process specified by the process/channel number pair pointed to by pointer into the three word buffer buffer. The contents of the buffer are dependent upon the device type of the kernel process. A value of 1 is returned by C.

SEE ALSO

getchar, putchar(a), setty(a)

DIAGNOSTICS

A value of -1 is returned from C if the kernel device driver is not a character device.

NAME

getcsw - get console switch register setting

SYNOPSIS

```
(getcsw = 46.)
getcsw()
```

DESCRIPTION

Getcsw returns the console switch register setting.

SEE ALSO

DIAGNOSTICS

NAME

growseg- increase or decrease the size of a segment

SYNOPSIS

```
(growseg = 8.)
growseg(segnum,size)
```

DESCRIPTION

The segment indexed by segnum is set to size words. New swap space will be allocated if the new size causes the segment to cross a sector boundary on the swap device. The new piece is zeroed and appears at the high address end for normal segments and the low address end for stack segments. If the segment is decreasing in size, the high address end is truncated.

ALSO SEE

DIAGNOSTICS

Return of -1:

1) segnum points to a null segment
2) segnum points to an inactive segment
3) System swap space is saturated
4) New size causes the process to exceed available memory (the "pcbnold" bit will be set on all user mode segments if fail return is for this reason).
5) The new size will cause the segment to require nonexistant or privileged segmentation registers (eg the new size of a segment starting in user I-space register 7 causes the segment to spill over into D-space base register 0).

Fault return with code 10.:

1) segnum exceeds the number of available segment table entries
2) the segment is non-swap

NAME

getmsg - get a message

SYNOPSIS

```
(getmsg = 30.)
getmsg(&msgbuf)
int *msgbuf;    /* pointer to message buffer */
```

DESCRIPTION

Getmsg gets a message in the buffer msgbuf from the process message input queue. (See sendmsg(a) for message header description). If no message is on this process's input queue, the process roadblocks within the library routine, waiting for a message event. The receiver of the message must fill in the mssize word to indicate the largest message he is willing to receive. The size of the message is exclusive of the size of the message header. Messages are dequeued according to FIFO.

SEE ALSO

sendmsg(a), sndmsgfrom(a), gettype(a).

DIAGNOSTICS

NAME

inhibit - run process at priority one

SYNOPSIS

```
(inhibit = 22.)
inhibit()
```

DESCRIPTION

Inhibit changes the hardware priority at which the process runs to one, to protect critical regions in the supervisor code from receiving events which may be received at processor priority one. The critical region can only be protected for up to 3 clock ticks (~48 msec.) before the processor priority is lowered to zero by the system. A process should use permit (see permit(a)) to lower the processor priority upon exiting the critical region.

SEE ALSO

permit(a).

DIAGNOSTICS

NAME

gettype - get a message of given type

SYNOPSIS

```
(gettype = 31.)
gettype(msgbuf)
int *msgbuf;    /* pointer to message buffer */
```

DESCRIPTION

Gettype gets a message of a particular type mstype in the buffer msgbuf from the process message input queue. (See sendmsg(a) for message header description). If no message is on this process's input queue of the type mstype the process roadblocks within the library routine, waiting for a message event (In assembly language, the c-bit is set). The receiver of the message must fill in the mssize word to indicate the largest message he is willing to receive. The size of the message is exclusive of the size of the message header. He must also fill in the message type mstype which he expects to receive back. This system function is provided to give the user control over the order in which he receives the different types of messages. Messages are dequeued according to FIFO.

SEE ALSO

sendmsg(a), sndmsgfrom(a), getmsg(a).

DIAGNOSTICS

NAME

lockseg - lock a segment in memory

SYNOPSIS

```
(lockseg = 11.)
lockseg(segnum)
```

DESCRIPTION

The segment indexed by segnum is locked in memory (eg can not be swapped or moved from its current position in physical memory). By convention all processes which issue I/O messages are expected to lock the segment in memory before issuing the I/O message. The kernel will unlock the segment when the I/O is complete.

ALSO SEE

alockseg(a), lockid(a), ulockseg(a), ulockid(a)

DIAGNOSTICS

C returns -1 if:

1) segnum points to a null segment table entry
2) segnum points to an inactive segment.

Fault return with code 10, if segnum points beyond the end of the segment table.

NAME

jobchg - change control to next process

SYNOPSIS

```
(jobchg = 23.)
jobchg()
```

DESCRIPTION

Jobchg causes the current process to give up control to the next process which is ready to run. If there is no other process ready to run, control will be returned back to the current process.

SEE ALSO

DIAGNOSTICS

A -1 is returned from C if no process is ready to run.

NAME

permit - run process at priority zero

SYNOPSIS

```
(permit = 47.)
permit()
```

DESCRIPTION

Permit changes the hardware priority at which the process runs to zero. A process should use permit to lower the processor priority upon exiting a critical region. The critical region must be protected by calling inhibit.

SEE ALSO

inhibit(a).

DIAGNOSTICS

NAME

lockid - increment the lock count on a segment

SYNOPSIS

```
(lockid = 12.)
lockid(segid)
```

DESCRIPTION

The lock count on the segment segid is incremented. Lockid differs from lockseg in two important respects: the segment must already be locked before lockid is called and segid need not be in the address space of the calling process. Processes initiating multiple independent I/O transfers into a segment belonging to another process find lockid useful.

ALSO SEE

alockseg(a), lockseg(a), ulockseg(a), ulockid(a)

DIAGNOSTICS

C returns -1 if:

1) segid is not a valid segment id
2) segid is not locked in memory.

NAME

pswap - remove non-swap status from a process

SYNOPSIS

```
(pswap = 17.)
pswap()
```

DESCRIPTION

Non-swap status is removed from all segments in the process segment table and the non-swap bit in the kernel scheduler tables is cleared. This is the opposite of punswap.

ALSO SEE

punswap(a)

DIAGNOSTICS

NAME

openseg - add a segment id to the process segment table

SYNOPSIS

```
(openseg = 1)
openseg(segnum,segid,segflags)
```

DESCRIPTION

The process segment table entry indexed by segnum is filled in with segflags and segid. The user count of the segment named by segid is incremented. This system fuction is used by the process manager in producing a core dump of a process.

ALSO SEE

alocseg(a), setmap(a), unblkseg(a)

DIAGNOSTICS

C returns -1 if segid is not a valid segment id or if segid is unallocated. Fault return with fault code 10 if the segment table entry indexed by segnum is non-zero. A return of zero if segid is in the blocked state.

NAME

putchar - output characters to character device driver

SYNOPSIS

```
(putchar = 193.)
putchar(pointer, segid, offset, count, &flag)
int *pointer;   /* pointer to process, channel pair */
int segid;      /* ID of segment being written from */
int offset;     /* byte offset into segment */
int count;      /* byte count */
int *flag;      /* flag word */
```

DESCRIPTION

Putchar outputs count characters from the caller from the segment specified by segid starting at offset bytes into the segment. If the segment is a stack segment the offset is from the end of the segment. Pointer points to the pair of words specifying the process number of the kernel process and the logical channel number of the device controlled by the kernel process. The call to the kernel within the library routine returns the byte count. If all bytes are not written in one call to the kernel device driver, the library routine roadblocks, waiting for a wakeup from the kernel driver process. In the roadblock state, if a quit or interrupt signal is received, the user is unroadblocked, and the non-zero value of the flag causes an immediate return to the user. As many calls are issued to the kernel driver process as required to write out all count bytes. The total bytes written are returned from the library routine.

SEE ALSO

getchar(a), setty(a), getty(a)

DIAGNOSTICS

If an error condition is detected (illegal buffer address or size) the negative value of the error code is returned in C.

NAME

pstart - start process

SYNOPSIS

```
(pstart = 42.)
pstart(pprior, chan, segid, param, parent)
int pprior;     /* processor priority (1 <= pprior <= 7) */
int chan;       /* process control channel number */
int segid;      /* segment ID */
int param;
int parent;     /* process number of parent */
```

DESCRIPTION

Pstart puts an entry for the new process in the DCT table. The new process must be started by the parent process by sending it a wakeup event. The processor priority specified pprior is 1 for a supervisory process and from 3 to 7 for a kernel process. The priority of 2 is not allowed. The process control channel is specified by chan. For a supervisory process, segid is the segment ID of the PCB of the process being started. For a kernel process, segid is the segment ID of the first kernel process segment. The basic priority at which a process is to run (0 - 0360) is specified by param for a supervisory process. For a kernel process, param specifies the total number of segments in the process. The process number of the started process is returned from C. The high order byte of the process number is the incarnation count and the low order byte is the entry number in the DCT table.

SEE ALSO

DIAGNOSTICS

A -1 is returned from C if the process could not be started.

NAME

rti - return from trap

SYNOPSIS

```
(rti = 37.)
rti(pc, ps)
int pc;
int ps;
```

DESCRIPTION

Rti returns back up to the user at the address specified by pc and the hardware priority specified by ps.

SEE ALSO

DIAGNOSTICS

NAME

punswap - make a process non-swap

SYNOPSIS

```
(punswap = 16.)
punswap()
```

DESCRIPTION

All the active segments in the process segment table are made non-swap (segments are not "locked" since they can still be shifted in memory to reduce fragmentation). The non-swap bit is set in the kernel process tables which permits the scheduler to dispatch to the process without calling the memory manager to check if all the segments are in memory. This is useful for processes which require service within real time limits shorter than the swap time.

ALSO SEE

pswap(a)

DIAGNOSTICS

C returns -1 if the memory available for swapping will be less than 8K words after all the segments are made non-swap.

NAME

sendmsg - send a message

SYNOPSIS

```
(sendmsg = 28.)
sendmsg(msgbuf)
int *msgbuf;    /* pointer to message */
```

DESCRIPTION

Sendmsg sends a message from the current process to another process (kernel or supervisor-user type). The message to be sent starts at msgbuf and may be up to 112 words long. A message consists of a six word header defined by the following structure:

```
struct msghdr {
        int     *mslink;        /* pointer to next input message */
        int     msfrom;         /* sending process number */
        int     msto;           /* receiving process number */
        char    mssize;         /* message size in words */
        char    mstype;         /* message type */
        int     msident;        /* message identifier */
        char    msstat;         /* message status word */
        char    msseqnum;       /* message sequence number */
};
```

and the sender's data. The sender need only fill in the msto, mstype and mssize fields of the message. The mssize word is the size of the sender's message in words exclusive of the header. The mstype byte may be any number from 0 to 0376. The value of 0377 is reserved for acknowledgement messages. The sender may fill in msident in order to identify a particular acknowledgement message, as this word is never modified during the life of this message. The message is verified and copied into a kernel address space message buffer area. Here the msfrom word is filled in by the kernel as well as the message sequence number. The message is put on the input queue of messages for the msto process using the mslink word. A programmed interrupt request is enabled by sending a message event to the msto process. The message sequence number msseqnum is used only for debugging purposes. The msstat byte is filled in by the receiver of this message in its acknowledgement to this message. It contains the error code if non-zero. The value of -1 is reserved by the system for the case where the intended receiver process does not exist or is aborted abnormally.

SEE ALSO

sndmsgfrom(a), getmsg(a), gettype(a).

DIAGNOSTICS

If the message is too big, a -1 is returned from C.

NAME

rmovseg - remove a segment from a process virtual address space

SYNOPSIS

```
(rmovseg = 5.)
rmovseg(segnum)
```

DESCRIPTION

The segment indexed by segnum is removed from the process virtual address space (the "active" bit is cleared in the segment table status word). If the "non-swap" bit of the segment table status word is non-zero, the bit is cleared and the segment "non-swap" count is decremented. The segment activity count is decremented and if zero, the segment becomes a candidate for swapping. This system function should be used if the contents of the segment are to be used at a later time. If the segment is used for scratch storage and the current data are not needed when the segment is reactivated, dropseg should be used.

ALSO SEE

dropseg(a), freeseg(a).

DIAGNOSTICS

C returns -1 if the segnum points to a null segment table entry. Fault return with code 10 if segnum exceeds the available number of segment table entries.

DIAGNOSTICS

If the message is too big, the port number is invalid or the receiving process is unable to hold any more messages on its input queue, a -1 is returned from C.

NAME

sendport - send message through port

SYNOPSIS

```
(sendport = 49.)
sendport(msgbuf)
int *msgbuf;     /* pointer to message */
```

DESCRIPTION

Sendport sends a message from the current process to the process connected to the specified port number. The message to be sent starts at msgbuf and may be up to 112 words long. A message consists of a six word header defined by the following structure:

```
struct msghdr {
        int     *mslink;        /* pointer to next input message */
        int     msfrom;         /* sending process number */
        int     msto;           /* receiving process number(conn. t
        char    mssize;         /* message size in words */
        char    mstype;         /* message type */
        int     msident;        /* message identifier */
        char    msstat;         /* message status word.*/
        char    msseqnum;       /* message sequence number */
};
```

and the sender's data. The sender need only fill in the msto, mstype and mssize fields of the message. The mssize word is the size of the sender's message in words exclusive of the header. The mstype byte may be any number from 0 to 0376. The value of 0377 is reserved for acknowledgement messages. The msto field must contain the port number to which the receiving process is connected. The sender may fill in msident in order to identify a particular acknowledgement message, as this word is never modified during the life of this message. The message is verified and copied into a kernel address space message buffer area. Here the msfrom word is filled in by the kernel as well as the message sequence number. The message is put on the input queue of messages for the process connected to the port number specified in msto using the mslink word. A programmed interrupt request is enabled by sending a message event to the receiving process. The message sequence number msseqnum is used only for debugging purposes. The msstat byte is filled in by the receiver of this message in its acknowledgement to this message. It contains the error code if non-zero. The value of -1 is reserved by the system for the case where the intended receiver process does not exist or is aborted abnormally.

SEE ALSO

sendmsg(a), sndmsgfrom(a), getmsg(a), gettype(a).

NAME

setmap - set access, mode and starting segmentation register

SYNOPSIS

```
(setmap = 9.)
setmap(segnum,access,basereg)
```

DESCRIPTION

Setmap sets the virtual address space (supervisor or user), virtual address in I or D space, and the access permissions including expansion direction, of the segment indexed by segnum. The virtual address space and access are specified by access as follows:

bit 13:12	mode (1 - supervisor, 2 - user)
bit 5	1 if segment is shareable
bit 3	expansion direction (1 for stack segment)
bit 2:0	access (6 - read/write, 2 - read only)

The virtual address is specifed by basereg:

bit 3	I or D space (1 for D space)
bit 2:0	starting segmentation register

Setmap does not affect the activity of a segment (eg if the segment was inactive, it remains inactive).

ALSO SEE

addseg(a)

DIAGNOSTICS

C returns -1 if:

1) segnum points to a null entry
2) basereg causes the segment to spill over into non-existent segmentation registers

Fault return with code 10 if segnum exceeds the available number of segment table entries.

NAME

setdspac - set user-supervisor d-space bits

SYNOPSIS

```
(setdspac = 45.)
setdspac(dspacbits)
int dspacbits;
```

DESCRIPTION

Setdspac sets the d-space bits for user and supervisor address spaces according to dspacbits. Bit 0 = 1 turns on user d-space and bit 1 = 1 turns on supervisor d-space.

SEE ALSO

DIAGNOSTICS

NAME

setty - set state of tty driver process

SYNOPSIS

```
(setty = 194.)
setty(pointer, buffer, &flag)
int *pointer;   /* pointer to process, channel pair */
int buffer[3];  /* 3 word buffer */
int *flag;      /* pointer to event flag */
```

DESCRIPTION

Setty sets the state of the tty kernel driver process specified by the process/channel number pair pointed to by pointer from the three word buffer buffer. The contents of the buffer are dependent upon the device type of the kernel process. If the kernel driver still has characters on its output queue, the library routine roadblocks the user, waiting for a wakeup event before returning back to the kernel driver. In the roadblock state, if a quit or interrupt signal is received, the user is unroadblocked, and the a nonzero value of the flag causes an immediate return to the user with the error condition set. A value of 1 is returned by C normally.

SEE ALSO

getchar, putchar(a), getty(a)

DIAGNOSTICS

A value of -1 is returned from C if the kernel device driver is not a character device.

NAME

setime - set time

SYNOPSIS

```
(setime = 38.)
setime(time)
int time[2];
```

DESCRIPTION

Setime sets the time in the kernel according to the 32-bit time in the array time.

SEE ALSO

getime(a).

DIAGNOSTICS

NAME

sleep - set a bit pattern to sleep on

SYNOPSIS

```
(sleep = 26.)
sleep(pattern)
```

DESCRIPTION

Sleep sets the bit pattern pattern in the current process's DCT entry. Then when this process is subsequently road-blocked, it is re-scheduled by a call to wakeup (pattern).

SEE ALSO

wakeup(a), rdblk(a), psleep(b).

DIAGNOSTICS

If a sleep is impossible, a -1 is returned from C.

NAME

setprior - set priority of process

SYNOPSIS

```
(setprior = 41.)
setprior(priority)
int priority;
```

DESCRIPTION

Setprior sets the priority of the process to priority (0 - 0277). This is the base priority of the process set in the PCB word p_prior and the process DCT table entry, and is the priority at which the process normally runs.

SEE ALSO

DIAGNOSTICS

NAME

spacaloc - allocate memory for a segment; add the segment to the process virtual address space

SYNOPSIS

```
(spacaloc = 7)
spacaloc(segnum)
```

DESCRIPTION

Memory is allocated, zeroed, and assigned to the segment indexed by segnum. The segment is then added to the process virtual address space (see addseg(a)). If the calling process does not have write permission on the segment, this system function reduces to addseg.

ALSO SEE

addseg(a)

DIAGNOSTICS

C returns -1 if the segnum points to a null segment table entry. Fault return with code 10 if segnum exceeds the available number of segment table entries.

NAME

sizeseg - get size of segment

SYNOPSIS

```
(sizeseg = 20.)
size = sizeseg(segnum)
```

DESCRIPTION

Sizeseg returns the size of the segment (in words) indexed by segnum in the process segment table in the PCB.

SEE ALSO

DIAGNOSTICS

From C, a -1 is returned if the segment does not exist, i.e. there is no entry in the process's PCB.

NAME

sswap - remove non-swap status from a segment

SYNOPSIS

(sswap = 19.)
sswap(segnum)

DESCRIPTION

The non-swap bit in the process segment table indexed by segnum is cleared and the segment non-swap count is decremented (if zero the segment can be swapped out of memory). The process non-swap status in the scheduler tables is also cleared.

ALSO SEE

punswap(a), pswap(a), sunswap(a)

DIAGNOSTICS

C returns -1 if the segnum points to a null segment table entry. Fault return with code 10 if segnum exceeds the available number of segment table entries.

NAME

sndmsgfrom - send a message from a process

SYNOPSIS

```
(sndmsgfrom = 29.)
sndmsgfrom(msgbuf)
int *msgbuf;    /* pointer to message */
```

DESCRIPTION

Sndmsgfrom sends a message from the current process to another process (kernel or supervisor-user type) on behalf of the process which sent a message to the current process (See sendmsg(a) for message header description). It differs from sendmsg only in that the msfrom word in the message header must also be filled in by the sender. This word is not touched by the kernel emt trap handler. The net result is that the acknowledgement to this message is sent directly back to the original sender. Tnis function is provided particularly for a supervisor-mode file manager process to send read/write messages directly to the device driver processes which in turn send the acknowledgement messages directly back to the original sender in order to reduce message traffic. The msstat byte is filled in by the receiver of this message in its acknowledgement to this message. It contains the error code if non-zero. The value of -1 is reserved by the system for the case where the intended receiver process does not exist or is aborted abnormally.

SEE ALSO

sendmsg(a), getmsg(a), gettype(a).

DIAGNOSTICS

If the message is too big, a -1 is returned from C.

NAME

sysproc - system process

SYNOPSIS

```
(sysproc = 48.)
sysproc(portnum, flag)
int portnum;    /* system port number */
int flag;       /* function flag */
```

DESCRIPTION

Sysproc performs some operation on one of the system process ports specified by portnum according to the value of the function flag. Portnum is a value from 0 to one less than the maximum process port numbers (currently 4). For flag equal to 0, the current process is connected to the process port portnum. For flag equal to 1, the current process is disconnected from the process port. For flag equal to 2, the value of the process connected to the process port portnum is returned.

SEE ALSO

sendport(a).

DIAGNOSTICS

C returns -1 if the portnum is invalid or the function flag is invalid.

NAME

sunswap - make a segment non-swap

SYNOPSIS

```
(sunswap = 18.)
sunswap(segnum)
```

DESCRIPTION

The segment indexed by segnum is made non-swap (the segment is not locked since it can still be shifted in memory to reduce fragmentation).

ALSO SEE

punswap(a), pswap(a), sswap(a)

DIAGNOSTICS

C returns -1:

1) segnum points to a null segment
2) segnum points to an inactive segment
3) memory available for swapping will be less than 8K words after the segment is made non-swap

Fault return with code 10 if segnum points beyond the end of the segment table.

NAME

ulockid - decrement the lock count on a segment

SYNOPSIS

```
(ulockid = 15.)
ulockid(segid)
```

DESCRIPTION

The lock count on the segment segid is decremented.

ALSO SEE

alockseg(a), lockseg(a), lockid(a), ulockseg(a)

DIAGNOSTICS

C returns -1 if segid is not a valid segment id. Fault return with code 10 if segid is not locked in memory.

NAME

toutset - set time-out

SYNOPSIS

```
(toutset = 40.)
toutset(ticks)
```

DESCRIPTION

Toutset schedules a time-out event to occur ticks 60th's of a second later. A value of ticks equal to zero disables the time-out event. This system function is used for "profiling" in UNIX and for putting the process asleep for a fixed time. It may also be used to schedule a task (process) at fixed time intervals.

SEE ALSO

event(a), clrevent(a), enevent(a).

DIAGNOSTICS

NAME

unblkseg - unblock a named segment

SYNOPSIS

```
(unblkseg = 2)
unblkseg(segnum)
```

DESCRIPTION

The segment id indexed by segnum is unblocked. All processes sleeping on the segment id are awakened.

ALSO SEE

alocseg(a), openseg(a)

DIAGNOSTICS

C returns -1 if the segment table entry indexed by segnum is zero. Fault return with fault code 10 if segnum is greater than the number of available segment entries in the PCB segment table.

NAME

ulockseg - decrement the lock count on a segment

SYNOPSIS

```
(ulockseg = 14.)
ulockseg(segnum)
```

DESCRIPTION

The lock count on the segment indexed by segnum is decremented. Ulockseg is used primarily by the process manager to remove kernel processes from memory.

ALSO

alockseg(a), lockseg(a), ulockid(a)

DIAGNOSTICS

C returns -1 if the segnum points to a null segment table entry. Fault return with code 10 if segnum exceeds the available number of segment table entries.

NAME

writeseg - Force a segment to be written back

SYNOPSIS

```
(writeseg = 10.)
writeseg(segnum)
```

DESCRIPTION

The writeback bit is set on the segment id indexed by segnum.

ALSO SEE

DIAGNOSTICS

C returns -1 if the segnum points to a null segment table entry. Fault return with code 10 if segnum exceeds the available number of segment table entries.

NAME

wakeup - wakeup all processes sleeping on a pattern

SYNOPSIS

```
(wakeup = 27.)
wakeup(pattern)
```

DESCRIPTION

Wakeup wakes up all processes sleeping on the bit pattern pattern. A succesful call returns the value of 1.

SEE ALSO

sleep(a), rdblk(a)

DIAGNOSTICS

INTRODUCTION TO KERNEL EMT CALLS

Section B of this manual lists all the kernel EMT entries into the kernel from the kernel mode processes. In most cases two calling sequences are specified, one of which is usable from assembly language, and the other from C. Most of these calls have an error return. From assembly language an erroneous call is always indicated by turning on the c-bit of the condition codes. The presence of an error is most easily tested by the instructions bes and bec (``branch on error set (or clear)''). These are synonyms for the bcs and bcc instructions.

From C, an error condition is indicated by an otherwise impossible returned value. Almost always this is -1; the individual sections specify the details.

A kernel process is defined in the kernel in a DCT (dispatcher control table) entry. The structure of a DCT entry is:

```
struct dct {
  int     flags;  /* process flag bits */
  int     flink;  /* pointer to next process on this queue at
                     hardware this priority */
  int     sleep;  /* sleep bit pattern */
  int     *mesg;  /* pointer to first message on input queue
                     */
  int     time;   /* time-out value in 60ths of a second */
  int     eflags; /* event flags */
  int     process;/* process number */
  char    cchan;  /* control channel */
  char    status; /* process status bits */
  int     segadr; /* physical block address of start of code
                     for kernel process, process control block
                     segment ID for supervisor-user process */
  int     prior;  /* priority bytes */
  char    mcount; /* count of messages on process input queue
                     */
  char    fill;   /* not used */
}
```

The variable process is externally defined in a kernel mode process which references it. A kernel mode process runs at its specified hardware priority in flags and uses I-space only to provide protection for the kernel tables in D-space. The code for a kernel process must begin at address 060000 and may be up to 12K words long. Kernel base register 3 (KBR3) is set up to the beginning of the code. KBR4 and KBR5 are only set up if required. However KBR5 is used by the iomap emt call to set up a transfer directly into a supervisor or user segment for character I/O. KBR6 is always set up for access to the kernel library routines, such as the general tty routines. A kernel process uses the kernel stack. It also has access to the I/O address space by means of KBR7 and to the kernel message buffers by means of KBR0.

Each kernel process has a 27 word header preceeding the actual code with the following structure:

```
struct k_pcb {
        struct {
                int sar;/* segment address register */
                int sdr;/* segment descriptor register */
        } kr[6];        /* kernel base register settings for
                           KBR3, KBR4, KBR5 (I and D) */
        int     k_segid[6];     /* segment ID's (I and D) */
        int     k_sav;  /* pointer to base register save
                           routine in kernel */
        int     k_ent[3];/* process entry points:
                                   event entry point
                                   emt entry point
                                   fault entry point
                           */
        int     k_pn;   /* kernel process number */
        int     k_iflg; /* interrupt flag set by kernel when
                           interrupt occurs for this process */
        int     k_rsv[4];/* register save area upon entry to
                           this kernel process */
}
```

The kernel process is dispatched to by means of an interrupt or by the PIR (programmed interrupt request) triggered by an event. A kernel process does not need to be attached to an interrupt vector. In this case the process is dispatched to by means of an EMT call from the supervisor (EMT code >= 192.). Upon entry to the kernel process, the kernel saves the current settings of KBR3 and of KBR4 and KBR5 only if used. The base registers are restored on exit.

The kernel processes as well as the supervisor processes communicate via messages. However, kernel processes may read and write the message buffer area directly. EMT trap calls are provided to allocate and free messages in the kernel buffer area. The message consists of a six word header and up to 106 words of data. The size of the message may be a multiple of 16 words up to a total of 112 words. The layout of the message is defined by the structure:

```
struct {
        int     *mslink;/* link to next message on input queue
                           */
        int     msfrom; /* process from which message was re-
                           ceived */
        int     msto;   /* process to which message is to be
                           sent */
        char    mssize; /* bits 0-2     size in multiples of
                           16 words
                                   bit 3   allocated bit
                                   bit 4   acknowledgement bit
                                   bit 5   iolock bit
        char    mstype; /* message type */
        int     msident;/* message identification word only
                           used by sender */
```

```
        char    msstat;/* status byte set by receiver in  ack-
                          nowledgement  message or by system if
                          receiver process does not exist */
        char    msseqnum;/* message sequence number */
    }
```

The data in the body of the message must be filled in directly by the sender of the message.

NAME

alocmsg - allocate message buffer

SYNOPSIS

```
(alocmsg = 12.)
alocmsg(nwords)
int nwords;      /* number of words */
```

DESCRIPTION

Alocmsg allocates a contiguous piece of kernel memory in the message buffer pool. The size of message allocated is a multiple of 16 words and equal to or greater than nwords plus the size of the message header msghdr. The allocate bit and message size bits are set in the mssize byte of the message header. The rest of the message buffer is zeroed out. A pointer to the beginning of the message is returned in C.

In assembly language, r0 must contain the size of message buffer required in words. A pointer to the message buffer is returned in r0.

SEE ALSO

queuem(b), messink(b), dequeuem(b), freemsg(b), dqtype(b), queuemn(b)

DIAGNOSTICS

A 0 is returned from C if no message buffer space exists or if the size of message requested is greater than 112 words minus the message header size (6 words).

In assembly language, the c-bit is set to indicate an error.

NAME

dequeuem - dequeue a message

SYNOPSIS

```
(dequeuem = 9.)
dequeuem(process)
int process;    /* process number */
```

DESCRIPTION

Dequeuem dequeues a message from the input message queue for the process specified by process. The first message on the queue is removed first. A pointer to the message buffer is returned from C.

In assembly language, r1 must contain process. A pointer to the beginning of the message is returned in r0.

SEE ALSO

alocmsg(b), messink(b), queuem(b), freemsg(b), dqtype(b), queuemn(b)

DIAGNOSTICS

A null pointer is returned from C if there is no message on the input queue.

In assembly language, a null pointer is returned in r0.

NAME

atchintr - attach a process to an interrupt

SYNOPSIS

```
(atchintr = 17.)
atchintr(process, vector, entry)
int process;    /* process number */
int vector;     /* device vector address */
int entry;      /* process entry point */
```

DESCRIPTION

Atchintr attaches the process process to the interrupt vector at address vector. The PC,PS pair is loaded with the entry point entry and the priority of the process, respectively. This EMT trap is provided to enable the catching of interrupts for a device. If a process is not attached to a device, interrupts for this device are ignored. A value of 1 is returned from C.

In assembly language, the following registers must be set up:

r0	vector address
r1	process number
r2	entry point

The c-bit is cleared for a normal return.

SEE ALSO

dtchintr(b), attach(a), detach(a).

DIAGNOSTICS

A -1 is returned from C if the process does not exist, the vector address is out of range, or if access to the control and status register of the device produces a bus error.

In assembly language, the c-bit is set to indicate an error.

NAME

dtchintr - detach a process from an interrupt

SYNOPSIS

```
(dtchintr = 18.)
dtchintr(process, vector)
int process;    /* process number */
int vector;     /* device vector address */
```

DESCRIPTION

Dtchintr detaches the process process from the interrupt vector at address vector. The entry point for the process is cleared as well as the device control register. This EMT trap is provided to enable the disabling of interrupts for a device. If a process is not attached to a device, interrupts for this device are ignored. A value of 1 is returned from C.

In assembly language, the following registers must be set up:

r0	vector address
r1	process number

The c-bit is cleared for a normal return.

SEE ALSO

atchintr(b), attach(a), detach(a).

DIAGNOSTICS

A -1 is returned from C if the process does not exist or the vector address is out of range.

In assembly language, the c-bit is set to indicate an error.

NAME

dqtype - dequeue a particular message type

SYNOPSIS

```
(dqtype = 10.)
dqtype(process, type)
int process;    /* process number */
int type;       /* message type */
```

DESCRIPTION

Dqtype dequeues a message of type type from the input message queue for the process specified by process. The first message of this type on the queue is removed first. A pointer to the message buffer is returned from C.

In assembly language, r1 must contain the process number and r0 the message type. A pointer to the beginning of the message is returned in r0.

SEE ALSO

alocmsg(b), messink(b), dequeuem(b), freemsg(b), queuem(b), queuemn(b)

DIAGNOSTICS

A null pointer is returned from C if there is no message of this type on the input queue.

In assembly language, a null pointer is returned in r0.

NAME

getarg - get argument from SUP address space

SYNOPSIS

```
getarg(address)
int address;    /* address in supervisor D-space */
```

DESCRIPTION

Getarg gets the value of the argument pointed to by address in supervisor D-space. This function involves no EMT trap. The value is returned from C.

This function need not be called from assembly language.

SEE ALSO

putarg(b).

DIAGNOSTICS

NAME

freemsg - free up message buffer

SYNOPSIS

```
(freemsg = 11.)
freemsg(&msgbuf)
int *msgbuf;    /* pointer to message buffer */
```

DESCRIPTION

Freemsg frees up a message in the kernel message buffer pool pointed to by msgbuf. This is done by simply clearing the allocate bit in the status byte of the message header.

In assembly language, r0 must contain a pointer to the message.

SEE ALSO

alocmsg(b), messink(b), dequeuem(b), queuem(b), dqtype(b), queuemn(b)

DIAGNOSTICS

NAME

iolock - lock segment for I/O

SYNOPSIS

```
(iolock = 4.)
iolock(segid)
int segid;      /* segment ID */
```

DESCRIPTION

Iolock locks the segment named by segid into memory for an I/O transfer. This is done by incrementing a lock count associated with this segment in the segment descriptor table. The starting block address of this segment is returned from C.

In assembly language, r0 must contain the segment ID, segid. The starting block address of this segment is returned in r0.

SEE ALSO

uniolock(b).

DIAGNOSTICS

A -1 is returned from C if segid is not a valid segment ID or if the segment is not in memory.

In assembly language, the c-bit is set.

NAME

getime - get system time

SYNOPSIS

```
(getime = 21.)
getime(&tbuf)
int tbuf[2];    /* time buffer */
```

DESCRIPTION

Getime returns the 32-bit quantity specifying the time of the system with the high order time bits in first word and low order time bits in second word of buffer tbuf. The time specified by tbuf is in 60ths of a second. C returns a value of 1.

SEE ALSO

setime(b).

DIAGNOSTICS

NAME

ioqueuem - send message to I/O device driver

SYNOPSIS

```
(ioqueuem = 1.)
ioqueuem(&msgbuf)
int *msgbuf;    /* pointer to message buffer */
```

DESCRIPTION

Ioqueuem computes the physical address for an I/O transfer as defined by the message pointed to by msgbuf. The message body must contain:

word 0 segment id
word 1 word offset into segment
word 2 I/O byte count.

A negative word offset indicates offset is from top of segment. Upon return word 1 of the message will contain the 16 least significant bits of the physical address. The user status byte in msstat of the message contains the extension bits. This routine also sets the iolock bit in the mssize byte so that the segment will be unlocked by messink. A value of 1 is returned from C.

In assembly language, r0 must contain the message buffer address.

SEE ALSO

iomap(b), messink(b), iomsg(c).

DIAGNOSTICS

A null value is returned from C if the segment ID is not valid, if the segment is not in memory, if the segment is not locked for I/O, or if the transfer to be initiated would be outside the address space of the segment.

In assembly language, the c-bit is set.

NAME

iomap - map segid/offset to virtual address

SYNOPSIS

```
(iomap = 2.)
iomap(segid, offset, count)
int segid;      /* segment ID */
int offset;     /* byte offset into segment */
int count;      /* byte count */
```

DESCRIPTION

Iomap maps the virtual address given by segid and offset into a virtual address using kernel I-space register 5. If the segment is a stack segment, the byte offset is from the top of the segment. The virtual address is returned from C. Up to 8192 bytes may be transferred once this virtual address is determined.

In assembly language, r0 must contain the segment ID, segid, r1 the byte offset into the segment, and r2 the actual number of bytes to be transferred. The virtual address is returned in r0.

SEE ALSO

ioqueuem(b).

DIAGNOSTICS

A -1 is returned from C if segid is not a valid segment ID, if the segment is not in memory, or if the transfer to be initiated would be outside the address space of the segment.

In assembly language, the c-bit is set.

NAME

psignal - send events to processes on a control channel

SYNOPSIS

```
(psignal = 20.)
psignal(channel, evflags)
int channel;    /* control channel number */
int evflags;    /* event flags */
```

DESCRIPTION

Psignal sends the events specified by the bits set in evflags to all processes which have the control channel channel. Sending an event causes the system to set the appropriate bit in the process' control table and trigger the programmed interrupt at the processor priority of the receiving process. The processes are placed on the run queue at the highest possible software priority. Nothing is returned from C.

In assembly language, r0 must contain the channel number and r1 the event flags.

SEE ALSO

event(a), sendevent(b).

DIAGNOSTICS

NAME

messink - return a message

SYNOPSIS

```
(messink = 8.)
messink(&msgbuf)
int *msgbuf;    /* pointer to message buffer */
```

DESCRIPTION

Messink returns a message to the kernel. If the noack bit is on in the mssize byte of the message header, the message buffer is freed up for future allocation. Otherwise the message is returned to the original sender as an acknowledgement message (mstype = -1). The iolock bit in the mssize byte of the message header is also checked to see if the segment into which I/O was done is to be unlocked. The message is queued on the original sender's message input queue and a message event is sent to this process. A value of 1 is returned from C.

In assembly language, r0 must contain the message buffer address.

SEE ALSO

alocmsg(b), queuem(b), dequeuem(b), freemsg(b), dqtype(b), queuemn(b)

DIAGNOSTICS

A value of 0 is returned from C if the original sender process no longer exists.

In assembly language, a value of 1 is returned.

NAME

ptimer - set time-out value for process

SYNOPSIS

```
(ptimer = 13.)
ptimer(process, time)
int process;    /* process number */
int time;       /* time-out value */
```

DESCRIPTION

Ptimer sets a timeout value in the process process' DCT table entry. The time specified by time is in 60ths of a second. C returns a value of 1.

In assembly language, r0 contains the timeout value and r1 the process number.

SEE ALSO

timleft(b).

DIAGNOSTICS

If the process specified is invalid a null value is returned from C.

In assembly language, the c-bit is set.

NAME

psleep - put process to sleep on bit pattern

SYNOPSIS

```
(psleep = 14.)
psleep(process, pattern)
int process;    /* process number */
int pattern;    /* bit pattern */
```

DESCRIPTION

Psleep puts the process process to sleep on the bit pattern pattern. This function is implemented by setting the pattern in the DCT table. Return is immediate and is usually followed by a rdblk at the higher level to become effective. The process is unroadblocked by a call to pwakeup with the bit pattern pattern. A value of 1 is returned from C.

In assembly language, r0 must contain the bit pattern and r1 the process number.

SEE ALSO

pwakeup(b), sleep(a), wakeup(a).

DIAGNOSTICS

In C, a null value is returned if the process does not exist.

In assembly language, the c-bit is set.

NAME

pwakeup - wake up processes sleeping on bit pattern

SYNOPSIS

```
(pwakeup = 15.)
pwakeup(pattern)
int pattern;    /* bit pattern */
```

DESCRIPTION

Pwakeup wakes up all processes sleeping on the bit pattern pattern. The processes to be woken up are sent a wakeup event and set to run at a high priority. A value of 1 is returned from C.

In assembly language, r0 must contain the bit pattern.

SEE ALSO

psleep(b), sleep(a), wakeup(a).

DIAGNOSTICS

NAME

putarg - put argument into SUP address space

SYNOPSIS

```
putarg(address, value)
int address;    /* address in supervisor D-space */
int value;      /* value to be moved */
```

DESCRIPTION

Putarg puts value into the supervisor D-space address specified by address. This function involves no EMT trap.

This function need not be called from assembly language.

SEE ALSO

getarg(b).

DIAGNOSTICS

NAME

queuemn - queue message with no acknowledgement expected

SYNOPSIS

```
(queuemn = 6.)
queuemn(msgbuf)
int *msgbuf;   /* pointer to message buffer */
```

DESCRIPTION

Queuemn queues the message pointed to by msgbuf on the input message queue of the process specified by msto in the message header with no acknowledgement to this message expected. The current value of the message sequence number is placed in msseqnum of the message header and then updated. A message event is sent to the msto process. A value of 1 is returned from C.

In assembly language, r0 must contain the message buffer address.

SEE ALSO

alocmsg(b), messink(b), dequeuem(b), freemsg(b), dqtype(b), queuem(b)

DIAGNOSTICS

A value of 0 is returned from C if the msto process is not a valid process number. If no more messages can be put on the receiver's input message queue, the message buffer is freed up.

In assembly language, the c-bit is set.

NAME

queuem - queue message on input queue

SYNOPSIS

```
(queuem = 7.)
queuem(msgbuf)
int *msgbuf;    /* pointer to message buffer */
```

DESCRIPTION

Queuem queues the message pointed to by msgbuf on the input message queue of the process specified by msto in the message header. The current value of the message sequence number is placed in msseqnum of the message header and then updated. A message event is sent to the msto process. A value of 1 is returned from C.

In assembly language, r0 must contain the message buffer address.

SEE ALSO

alocmsg(b), messink(b), dequeuem(b), freemsg(b), dqtype(b), queuemn(b)

DIAGNOSTICS

A value of 0 is returned from C if the msto process is not a valid process number. In this case an error status code of -1 is returned to the sender as an acknowledgement message. If no more messages can be put on receiver's input message queue, the message buffer is freed up.

In assembly language, the c-bit is set.

NAME

sendevent - send event to a process

SYNOPSIS

```
(sendevent = 19.)
sendevent(process, evflags)
int process;    /* process number */
int evflags;    /* event flags */
```

DESCRIPTION

Sendevent sends the events specified by the bits set in evflags to the process specified by process. Sixteen event types may be specified, eight of which have the following pre-defined meanings:

```
0100000 wakeup
0040000 timeout
0020000 message
0010000 hangup
0004000 interrupt
0002000 quit
0001000 abort
0000400 init
```

The other eight event types are user-definable. Sending an event causes the system to set the appropriate bit in the process' control table and trigger the programmed interrupt at the processor priority of the receiving process.

In assembly language, r0 must contain the event flags and r1 the process number.

SEE ALSO

event(a).

DIAGNOSTICS

If the process specified does not exist, a -1 is returned from C.

In assembly language, the c-bit is set.

NAME

riteback - set altered bit on a segment

SYNOPSIS

```
(riteback = 5.)
riteback(segid)
int segid;      /* segment ID */
```

DESCRIPTION

Riteback sets the altered bit on the segment specified by segid. A value of 1 is returned from C.

In assembly language, r0 must contain the value of segid.

SEE ALSO

DIAGNOSTICS

A null value is returned from C if the segment is not in memory.

In assembly language, the c-bit is set.

NAME

timleft - get time-out value for process

SYNOPSIS

```
(timleft = 16.)
timleft(process)
int process;    /* process number */
```

DESCRIPTION

Timleft returns the time left in 60ths of a second before the time-out event is to be triggered for the process process.

In assembly language, r1 contains the process number. The time-out value is returned in r0.

SEE ALSO

ptimer(b).

DIAGNOSTICS

If the process specified is invalid or time-out was not enabled for this process a null value is returned from C.

In assembly language, the c-bit is set.

NAME

setime - set system time

SYNOPSIS

```
(setime = 22.)
setime(&tbuf)
int tbuf[2];    /* time buffer */
```

DESCRIPTION

Setime sets the 32-bit quantity specifying the time of the system with the high order time bits in first word and low order time bits in second word of buffer tbuf. The time specified by tbuf is in 60ths of a second. C returns a value of 1.

SEE ALSO

getime(b).

DIAGNOSTICS

INTRODUCTION TO INTER-PROCESS MESSAGE FORMATS

Section C of this manual describes the formats of all of the messages used in inter-process communication. A message consists of a 6 word header and up to 106 words of data in the body of the message. Messages may be sent from a kernel process or a supervisor process to any other kernel or supervisor process. The message header is defined by the following structure:

```
struct msghdr {
        int     *mslink;/* link to next message on input queue
                        */
        int     msfrom; /* process from which message was re-
                        ceived */
        int     msto;   /* process to which message is to be
                        sent */
        char    mssize; /* bits 0-2     size in multiples of
                        16 words
                                bit 3   allocated bit
                                bit 4   acknowledgement bit
                                bit 5   iolock bit
        char    mstype; /* message type */
        int     msident;/* message identification word only
                        used by sender */
        char    msstat;/* status byte set either by sender of
                        message or by the kernel */
        char    msseqnum;/* message sequence number */
}
```

The data in the body of the message must be filled in directly by the sender of the message and varies with the message type as well as the receiving process. Normally the sender need only fill in the msto and mstype fields of the message header.

In the case of a kernel process "sender", an alocmsg(nwords) EMT call is required to alocate space for the message in the kernel message buffer pool. This call fills in the appropriate "size" and "allocate" bits in the mssize field of the message. The msfrom field must also be filled in by the "sender". The msident field may be filled in by the "sender" only if he wishes to recognize the acknowledgement to this message. The kernel process may then fill in the body of the message and send it to the intended receiver by means of the queuem or queuemn (no acknowledgement expected from receiving process) EMT calls. The kernel queues the message on the input queue of the receiving process using the mslink word. The kernel also fills in the msseqnum byte, which is used strictly as a debugging aid. The msstat field of the message is filled in by the receiving process to pass back error status. The value of -1 is reserved by the kernel to indicate that the receiving process does not exist or received an abnormal termination.

In the case of a supervisor process "sender", the mssize field must be filled in. Here the mssize field is now the number of words in the body of the message excluding the header. The com-

NAME

uniolock - unlock segment for I/O

SYNOPSIS

```
(uniolock = 3.)
uniolock(segid)
int segid;      /* segment ID */
```

DESCRIPTION

Uniolock unlocks the segment named by segid. The uniolock function is called after an I/O transfer has taken place, to or from this segment. This is done by decrementing a lock count associated with this segment in the segment descriptor table. This routine need only be called if messink is not called. A value of 1 is returned from C.

In assembly language, r0 must contain the segment ID, segid.

SEE ALSO

iolock(b), messink(b).

DIAGNOSTICS

A -1 is returned from C if segid is not a valid segment ID or if the segment is not in memory.

In assembly language, the c-bit is set.

INTRODUCTION TO FILE MANAGER MESSAGES

The types of messages which the file manager is programmed to accept are:

EXEC	1	open file for execution
FORK	2	increment count on open files
READ	3	read from file
WRITE	4	write to file
OPEN	5	open file
CLOSE	6	close file
NMCODE	7	get segment name
CREAT	8	create file
LINK	9	link to a file
UNLINK	10	remove link from file
MDATE	11	modify date of file
CHDIR	12	change directory
INIT	13	initialization message
MKNOD	14	make a node
CHMOD	15	change mode of file
CHOWN	16	change owner of file
SYNC	17	update file systems on secondary
STAT	18	get status of file
FSIZE	19	get size of file
FSTAT	20	get status of open file
SMOUNT	21	mount file system
SUMOUNT	22	unmount file system
MOVE	23	move file into contiguous area
ALLOC	24	allocate contiguous space for file
OPENI	25	open file by inode number

For messages sent to the file manager, the first two words of the body of the message must contain the current directory and the user and group ID's. The rest of the message contains various arguments depending on the message type. The structure of a message to the file manager is given by:

```
struct {
        struct  msghdr;         /* 6 word message header */
        int     fm_cdir;        /* current working directory *
        char    fm_uid;         /* user ID */
        char    fm_gid;         /* group ID */
        int     fm_arg[];       /* list of arguments */
}
```

The fm_cdir is actually a file descriptor pointing to the inode maintained by the file manager which is actually the message sender's current directory. This value may be zero if the current directory is to be the root directory. The fm_uid and fm_gid identifiers are used to determine the message sender's access privileges to particular files. For all messages, error codes are passed back to the sender in msstat. A system error is indicated by a -1 value of msstat. The meanings of the possible error codes from the file manager are the same as the standard UNIX error codes as explained in section II of this manual. If the message type sent is illegal, an error code is also returned. The

plete message is formed in a message buffer in the supervisor address space. This message is sent to the intended receiver using the "sendmsg" EMT call. The kernel allocates space for the message in the kernel message buffer pool and copies in the message from the supervisor address space, converting the mssize word to the appropriate bit field in so doing. The mslink and msseqnum words are filled in by the kernel as in the case of a kernel process. In addition the kernel fills in the msfrom word in the message header.

Upon sending a message, the receiver is sent a message event to inform it of the presence of a message on its input queue. In the case of a kernel process receiver, no copy of the message occurs; a "dequeuem" or "dqtype" EMT call returns a pointer to the message buffer, allowing the receiver to directly access the body of the message. In the case of a supervisor process, a call to getmsg or gettype will initiate the copy of the message from the kernel message buffer pool to the supervisor buffer provided by the receiver, freeing up the space used by this message in the kernel message buffer pool in the process. The receiver need only fill in the maximum size message which he expects to receive.

This section of the manual details all message types for the processes sending and receiving messages. In each case the 6 word header is identical. Only the contents of the body of the message varies. The processes discussed include:

File Manager
Process Manager
Memory Manager
Scheduler
I/O Processes

Message types are discussed according to what types of messages each process is willing to accept.

NAME
chdir - change working directory

SYNOPSIS
chdir = 12.

ARGUMENTS(input)
Ø - name byte offset into message
1 - pathname of file

VALUES(returned)
Ø - new working directory file descriptor

DESCRIPTION
Chdir causes the directory specified by the given pathname to become the new working directory.

SEE ALSO
chdir(I), chdir(II).

DIAGNOSTICS
An error status byte is returned if the given name is not that of a working directory or it is not readable.

additional error codes are:

63 EBADTPYE bad message type

In messages which require a file name to be specified, the file pathname (a null-terminated string) is copied into the body of the message. A pathname may be up to 64 characters long. The start of each pathname string is specified by a byte offset into the body of the message starting at fm_arg[0].

NAME
chdir - change working directory

SYNOPSIS
chdir = 12.

ARGUMENTS(input)
Ø - name byte offset into message
1 - pathname of file

VALUES(returned)
Ø - new working directory file descriptor

DESCRIPTION
Chdir causes the directory specified by the given pathname to become the new working directory.

SEE ALSO
chdir(I), chdir(II).

DIAGNOSTICS
An error status byte is returned if the given name is not that of a working directory or it is not readable.

NAME
chmod - change mode of file

SYNOPSIS
chmod = 15.

ARGUMENTS(input)
Ø - name byte offset into message
1 - mode
2 - file pathname

VALUES(returned)
none

DESCRIPTION
The file specified by the given file name has its mode changed to mode. Modes are described in chmod(II). Only the owner of a file or the super-user may change the mode.

SEE ALSO
chmod(I), chmod(II).

DIAGNOSTICS
An error status byte is returned if the given file name cannot be found or if the current user is neither the owner of the file nor the super-user.

NAME
chown - change owner of file

SYNOPSIS
chmod = 16.

ARGUMENTS(input)
0 - name byte offset into message
1 - owner (uid, gid)
2 - file pathname

VALUES(returned)
none

DESCRIPTION
The file whose name is given has its owner changed to owner (low byte is uid, high byte is gid). Only the super-user may change the owner of a file. Changing the owner of a file removes the set-user-ID protection bit unless it is done by the super user.

SEE ALSO
chown(I), chown(II), passwd(V).

DIAGNOSTICS
An error status byte is returned for illegal owner changes.

NAME
close - close a file

SYNOPSIS
close = 6.

ARGUMENTS(input)
Ø - file descriptor

VALUES(returned)
none

DESCRIPTION
A close message will close the file specified by the file descriptor in argument Ø. If the file is a normal file, any extra extents beyond the length of the file are freed up. For contiguous files, the extents beyond the current length of the file are not freed up. If the file is a special device file, a close message is sent to the device driver by the file manager. Then the appropriate terminate message is sent to the process message to decrement the user count on the device driver and remove the driver from memory if the user count is Ø.

SEE ALSO
creat(II), open(II), creat(c), open(c), falloc(c).

DIAGNOSTICS
An error status byte is returned if an unknown file descriptor is specified.

NAME
creat - creat a new file

SYNOPSIS
creat = 8.

ARGUMENTS(input)
Ø - name byte offset into message
1 - mode bits

VALUES(returned)
Ø - file descriptor
1 - file type

DESCRIPTION
Creat creates a new file or opens the existing file specified by the null-terminated pathname in the message. If the file did not exist, it is given mode mode (see chmod(II)). If the file did exist, its mode and owner remain unchanged but it is truncated to Ø length; however, its extents are not released. The file is opened for writing.

SEE ALSO
write(II), close(II), stat(II), write(c), close(c).

DIAGNOSTICS
An error status byte is returned if a needed directory is not searchable, the file does not exist and the directory in which it is to be created is not writable, the file does exist and is unwritable or the file is a directory.

NAME
exec - open file for execution

SYNOPSIS
exec = 1.

ARGUMENTS(input)
Ø - name byte offset into message
1 - pathname of file

VALUES(returned)
fm_uid (conditional)
fm_gid (conditional)
Ø - file descriptor.

DESCRIPTION
This message is sent by a UNIX supervisor to open a file for execution. The execute permission bits are checked. If the ISUID bit is set in the file inode, the uid of the file is returned in the acknowledgement message. The same is true for the ISGID bit.

SEE ALSO
open(II), open(c).

DIAGNOSTICS
An error status byte is returned if the file is not executable.

NAME

falloc - allocate contiguous space for a file

SYNOPSIS

falloc = 24.

ARGUMENTS(input)

0 - name byte offset into message
1 - mode
2,3 - size (in bytes)
4 - pathname of file

VALUES(returned)

0 - file descriptor
1 - file type
4 - number of contiguous blocks allocated

DESCRIPTION

Falloc creates a new file specified by the given pathname with mode bits set to mode (see chmod(II)). The contiguous bit is also set for this file. Contiguous file space is allocated on secondary to make room for the specified number of bytes. The file is opened for writing. When the file is closed, any extra blocks beyond the size of the file are not freed up as in the case of a normal file.

SEE ALSO

creat(II), chmod(II), creat(c), chmod(c).

DIAGNOSTICS

An error status byte is returned if the file already exists, a needed directory is not searchable, or the directory in which is the file is to be created is not writable.

NAME

fmove - move file into a contiguous area

SYNOPSIS

```
fmove = 23.
```

ARGUMENTS(input)

```
Ø - file system device number
1 - inode number of file
2 - flag
```

VALUES(returned)

none

DESCRIPTION

The file specified by the file system and the inode number is locked by the file manager for exclusive use, prohibiting access by other users. The extents of the super-block of the file system are searched for a contiguous area into which the file can be moved. The file, if it is smaller than a critical size, is then moved into this area and the file map is updated appropriately. If the value of flag is zero the file is moved only if it can be moved to a lower contiguous area. If flag is non-zero, the file will be moved into the lowest contiguous area, even if not lower than the area it currently occupies.

SEE ALSO

recon(d), fmove(d), fmove(f).

DIAGNOSTICS

An error status byte will be returned, if the user is not super-user, the file is a special file, the file no longer exists, the file is active or not enough contiguous space can be found.

NAME
fork - increment count on open files

SYNOPSIS
fork = 2.

ARGUMENTS(input)
0 - number of files
1 - open count
2 - file descriptor 1
...
n - file dsecriptor n

VALUES(returned)
none

DESCRIPTION
The FORK message is used to increment the open count on the list of files specified, by the increment specified. This increment may be positive or negative.

SEE ALSO

DIAGNOSTICS
An error status byte is returned if a bad descriptor is specified.

NAME
fsize - get size of file

SYNOPSIS
fsize = 19.

ARGUMENTS(input)
0 - file descriptor

VALUES(returned)
0 - high order word of 32-bit size
1 - low order word of 32-bit size

DESCRIPTION
The size of the file specified by the file descriptor is returned as a two-word value. This message type is typically used to get size of file when a seek beyond the end of the file is requested by a UNIX user.

SEE ALSO
stat(c), fstat(c).

DIAGNOSTICS

NAME
fstat - get status of open file

SYNOPSIS
fstat = 20.

ARGUMENTS(input)
0 - file descriptor

VALUES(returned)
0-17 - file status as described in stat(c).

DESCRIPTION
The status of the given file is returned in the acknowledgement message (18 words). Tnis message is identical to stat except that it operates on open files instead of files given by name. It is most often used by UNIX to get the status of the standard input and output files, whose names are unknown.

SEE ALSO
ls(I), stat(II), fstat(II), stat(C), fs(D).

DIAGNOSTICS
An error status byte is returned if the file descriptor is unknown.

NAME
init - initialize file system manager

SYNOPSIS
init = 13.

ARGUMENTS(input)
0 - root file system device (major/minor)
2 - root device process number

VALUES(returned)
none

DESCRIPTION
The INIT message is sent by the kernel when booting up the system to pass the root device code and the root process device driver number to the file manager.

SEE ALSO

DIAGNOSTICS

NAME
link - link to a file

SYNOPSIS
link = 9.

ARGUMENTS(input)
0 - byte offset to name1
1 - byte offset to name2
2 - pathname of name1
...
n - pathname of name2

VALUES(returned)
none

DESCRIPTION
A link to name1 is created. The link has the name name2. Either name may be an arbitrary pathname.

SEE ALSO
link(II), link(I), unlink(II).

DIAGNOSTICS
An error status byte is returned if name1 cannot be found, when name2 already exists, when the directory of name2 cannot be written, when an attempt is made to link to a directory by a user other than the super-user, or when an attempt is made to link to a file on another file system.

NAME
mdate - modify date of file

SYNOPSIS
mdate = 11.

ARGUMENTS(input)
Ø - name byte offset into message
1,2 - date (32 bits)
3 - pathname of file

VALUES(returned)
none

DESCRIPTION
The modified time of the file whose name is specified is set to the 32-bit time passed in the message. Only the super-user or owner of the file may set the modified time of the file.

SEE ALSO
time(II), mdate(II).

DIAGNOSTICS
An error status byte is returned if the user is neither the owner nor the super-user or if the file cannot be found.

NAME

mknod - make a directory or a special file

SYNOPSIS

mknod = 14.

ARGUMENTS(input)

0 - name byte offset into message
1 - mode
2 - device

VALUES(returned)

none

DESCRIPTION

Mknod creates a new file whose name is that of the given file. The mode of the new file (including directory and special file bits) is initialized from mode. The first physical address of the file (first extent) is initialized from device. This is zero for a normal or a directory file. For a special file, device specifies which special file. Mknod may only be invoked by the super-user (i.e. fm_uid = 0).

SEE ALSO

mkdir(I), mknod(II), mknod(VIII), fs(D).

DIAGNOSTICS

An error status byte is returned if the file already exists or if fm_uid is not 0.

NAME

mount - mount file system

SYNOPSIS

mount = 21.

ARGUMENTS(input)

0 - name offset into message of special file
1 - name offset into message of root file
2 - read/write flag rwflag
3 - pathname of special file
n - pathname of root file

VALUES(returned)

none

DESCRIPTION

Mount mounts a removable file system on the block-structured special file special. Thereafter, all references to the root file name will refer to the root file on the newly mounted file system. The root file must already exist. Its old contents are inaccessible while the file system is mounted. The rwflag determines whether the file system can be written on. If it is 0 writing is allowed, if non-zero no writing is done.

SEE ALSO

mount(II), mount(VIII), umount(II).

DIAGNOSTICS

An error status byte is returned if the special file is inaccessible or not an appropriate file, the root file does not exist, special is already mounted or there are already too many file systems mounted.

NAME
nmcode - get name code for segment

SYNOPSIS
nmcode = 7.

ARGUMENTS(input)
0 - file descriptor
1 - byte offset in file

VALUES(returned)
0,1 - unique 32-bit name for segment

DESCRIPTION
The nmcode message is used to get a unique 32-bit name for a segment which is to be loaded from a file. The name is derived from the location of the file on secondary storage and is used to allocate memory space for a named segment.

SEE ALSO
intro(a).

DIAGNOSTICS
An error status byte is returned if the file descriptor does not correspond to an open file.

NAME

open - open file

SYNOPSIS

open = 5.

ARGUMENTS(input)

Ø - name byte offset into message
1 - mode
2 - file pathname

VALUES(returned)

Ø - file descriptor
1 - file type
2 - process number (for device file only)
3 - channel number (for device file only)
4,5 - size of file

DESCRIPTION

The open message requests that the file specified by the pathname pointed to by the name offset in argument zero is opened for reading (mode = Ø), writing (mode = 1) or for both reading and writing (mode = 2). The returned file descriptor must be saved to be used for subsequent read's, write's and close's. The file type is returned in bits 3-5 of argument one. These bits correspond to bits 12-14 in mode bits of the inode (see fs(d)). If the file type is a device file, the file manager will send the appropriate messages to the process manager to load and lock in memory the appropriate device driver. An open message will also be sent to this device driver if one is expected by this device driver. In this case, arguments 2 and 3 will contain the process number of the device driver and the logical channel of this driver, respectively.

SEE ALSO

creat(II), read(II), write(II), close(II).
creat(c), read(c), write(c), close(c).

DIAGNOSTICS

An error status byte is returned if the file does not exist, if one of the necessary directories does not exist or is unreadable, or if the file is not readable(resp. writable). Also an error status byte is returned if the device driver can not be loaded by the process manager or if a bus error is generated by accessing the device control and status register.

NAME

openi - open file specified by inode number

SYNOPSIS

openi = 25.

ARGUMENTS(input)

0 - file system device code
1 - file inode number
2 - mode

VALUES(returned)

0 - file descriptor
1 - file type
2 - process number (for device file only)
3 - channel number (for device file only)
4,5 - size of file

DESCRIPTION.

The openi message requests that the file specified by the pathname pointed to by the name offset in argument zero is opened for reading (mode = 0), writing (mode = 1) or for both reading and writing (mode = 2). The returned file descriptor must be saved to be used for subsequent read's, write's and close's. The file type is returned in bits 3-5 of argument one. These bits correspond to bits 12-14 in mode bits of the inode (see fs(d)). If the file type is a device file, the file manager will send the appropriate messages to the process manager to load and lock in memory the appropriate device driver. An open message will also be sent to this device driver if one is expected by this device driver. In this case, arguments 2 and 3 will contain the process number of the device driver and the logical channel of this driver, respectively.

SEE ALSO

creat(II), read(II), write(II), close(II).
creat(c), read(c), write(c), close(c).

DIAGNOSTICS

An error status byte is returned if the file does not exist, if one of the necessary directories does not exist or is unreadable, or if the file is not readable(resp. writable). Also an error status byte is returned if the device driver can not be loaded by the process manager or if a bus error is generated by accessing the device control and status register.

NAME
read - read from file

SYNOPSIS
read = 3.

ARGUMENTS(input)
0 - segment ID
1 - word offset into segment
2 - byte count
3 - file descriptor
4 - block number in file.

VALUES(returned)
4 - bytes read

DESCRIPTION
Read messages must specify a starting block number in the file to be read from as well as a byte count. The File Manager converts the block number into a block number on the particular file system. The bytes read will not extend beyond the end of the file. If the word offset is negative, the offset is from the end of the segment. The sender of the message is required to lock the segment into which the I/O transfer is to be done in memory. If the I/O transfer can be done in one operation, the acknowledgement to this message is sent directly from the particular device driver. Otherwise, the read messages are sent off in parallel to the appropriate device driver and waited for by the file manager.

SEE ALSO
open(II), open(c).

DIAGNOSTICS
A 0 is returned if end of file has been reached. An error status byte is returned if a bad descriptor is given, the read would extend beyond the end of the segment or if physical I/O errors occur.

NAME

stat - get file status

SYNOPSIS

stat = 18.

ARGUMENTS(input)

0 - name byte offset into message
1 - file pathname

VALUES(returned)

0-17 - file status as described below

DESCRIPTION

The status of the given file is returned in the acknowledgement message (18 words). It is not necessary to have any permissions with respect to the file, but all directories leading to the file must be readable. Starting at fm_arg[0], the returned status of the file in the message is described by:

```
struct {
        char    minor;          /* minor device of i-node */
        char    major;          /* major device of i-node */
        int     inumber;        /* i-node number */
        int     flags;          /* see below */
        char    nlinks;         /* number of links to file */
        char    uid;            /* user ID of owner */
        char    gid;            /* group ID of owner */
        char    size0;          /* high byte of 24-bit size */
        char    *size1;         /* low word of 24-bit size */
        int     addr[8];        /* extents or device number */
        int     atime[2];       /* time of last access */
        int     mtime[2];       /* time of last modification *
};
```

The flags are as follows:

```
100000  i-node is allocated
070000  3-bit file type
    000000   plain file
    040000   directory
    020000   character-type special file
    060000   block-type special file
    070000   record-type special file
010000  contiguous file
004000  set user-ID on execution
002000  set group-ID on execution
000400  read (owner)
000200  write (owner)
000100  execute (owner)
000070  read, write, execute (group)
000007  read, write, execute (others)
```

SEE ALSO
ls(I), stat(II), fstat(II), fstat(C), fs(D).

DIAGNOSTICS
An error status byte is returned if the file cannot be found..

NAME

sync - update super-block

SYNOPSIS

sync = 17.

ARGUMENTS(input)

none

VALUES(returned)

none

DESCRIPTION

Sync causes all information in file manager buffers that has been modified, to be written out. This includes modified super-blocks, modified i-nodes and delayed block I/O. It should be used by programs which examine a file system, for example check, df. It must always be used before a boot of the system.

SEE ALSO

sync(II), sync(VIII).

DIAGNOSTICS

NAME

umount - dismount file system

SYNOPSIS

umount = 22.

ARGUMENTS(input)

0 - name byte offset into message of special file
1 - pathname of special file

VALUES(returned)

none

DESCRIPTION

Umount dismounts the special file special so that it no longer contains a removable file system. The file associated with the special file reverts to its ordinary interpretation.

SEE ALSO

mount(II), mount(VIII), umount(II), mount(c).

DIAGNOSTICS

An error status byte is returned if no file system was mounted on the special file or if there are still active files on the mounted file system.

NAME

unlink - remove directory entry

SYNOPSIS

```
unlink = 10.
```

ARGUMENTS(input)

0 - name byte offset into message
1 - pathname of file

VALUES(returned)

none

DESCRIPTION

Unlink removes the entry for the file specified from its directory. If this entry was the last link to the file, the extents of the file are freed and the file is destroyed. If however the file was open in any process, the actual destruction is delayed until it is closed, even though the directory entry has disappeared.

SEE ALSO

rm(I), rmdir(I), link(II), unlink(II).

DIAGNOSTICS

An error status byte is returned if the file does not exist or if its directory cannot be written. Write permisssion is not required on the file itself. Only the super-user may unlink a directory.

NAME

write - write to file

SYNOPSIS

write = 4.

ARGUMENTS(input)

Ø - segment ID
1 - word offset into segment
2 - byte count
3 - file descriptor
4 - block number in file.

VALUES(returned)

4 - bytes written

DESCRIPTION

Write messages must specify a starting block number in the file to be written to as well as a byte count. The File Manager converts the block number into a block number on the particular file system. If the word offset is negative, the offset is from the end of the segment. The sender of the message is required to lock the segment into which the I/O transfer is to be done in memory. If the I/O transfer can be done in one operation, the acknowledgement to this message is sent directly from the particular device driver. Otherwise, the write messages are sent off in parallel to the appropriate device driver and waited for by the file manager.

SEE ALSO

creat(II), open(II), creat(c), open(c), falloc(c).

DIAGNOSTICS

An error status byte is returned if a bad descriptor is given, the write would extend beyond the end of the segment or if physical I/O errors occur.

INTRODUCTION TO I/O PROCESS MESSAGES

Most block and record device I/O driver processes are programmed to accept messages of the following types:

```
IOREAD  1
IOWRITE 2
IOOPEN  3
IOCLOSE 4
```

However some I/O device drivers may not be programmed to accept IOOPEN and IOCLOSE messages, as they may be unnecessary for these devices. When the I/O device driver process is built using ldp(e) one of the specifications is whether the process will accept open and close messages. The format of a message to the device drivers is specified by the following structure:

```
struct io_msgd {
  struct msghdr io_dhr; /* message 6-word header */
  int    msiosid;  /* ID of segment into which or from
                      which I/O is to take place */
  int    msioba;   /* word offset into segment */
  int    msiobc;   /* number of bytes to be read
                      or written */
  char   msiodev;  /* logical device number */
  char   msiob0;   /* high order byte of block number */
  char   *msiob1;  /* low order word of block number */
  char   msiopri;  /* priority of I/O */
  char   msiotry;  /* number of retries on error */
  int    msiocnt;  /* number of bytes read */
  int    msfill1;  /* scratch word, may be used
                      by driver */
  int    msfill2;  /* scratch word, may be used
                      by driver */
  int    msfill3;  /* scratch word, may be used
                      by driver */
```

The IOOPEN and IOCLOSE messages need only specify the logical device number of the particular device driver and the number of times that the device driver file is open. This information is normally only available to the file manager process; the file manager thus will automatically send the open and close messages to the device drivers when the special files are opened and closed. The IOREAD and IOWRITE messages must specify a total of 5 arguments besides the normal parameters in the message header as discussed in the introduction. These are the ID of the segment into or from which the I/O transfer is to take place, the word offset into this segment, the number of bytes to be transferred, the logical device number of the driver and the block number at which the transfer is to start. The segment may be either in supervisor or user address space, but it must be locked for I/O by the sender of the message before the message is sent. The word offset into the segment is from the beginning of the segment if it is positive, otherwise it is from the end of the segment (e.g. for stack segment). Nor-

mally the file manager will make a call to the ioqueuem routine to determine if the transfer is within the bounds of the segment and if the segment is locked in memory. Ioqueuem then sends message to the appropriate driver. This routine will convert the virtual address given by msiosid and msioba into a 18-bit physical address. The lower 16 bits are returned in msioba and the upper 2 bits are set in the msustat byte of the message status byte. The total number of bytes read or written are returned in msiocnt. By invoking the messink routine when the I/O transfer is complete, the segment is unlocked.

SEE ALSO

ioqueuem(b), open(c), close(c).

DIAGNOSTICS

An error status byte is returned if the segment is not locked for I/O, the segment does not exist, the I/O transfer would extend beyond the end of the segment or if a physical I/O error occurs in the transfer.

INTRODUCTION TO THE PROCESS MANAGER MESSAGES

The message types accepted by the process manager are:

P_CREAT	1	Create a process from a pathname
P_DUMP	3	Produce a core dump of a process
P_INIT	4	Initialize the process manager (sent when the system is first booted)
P_WAIT	5	Sent to the process manager when a process it has created dies

In the following descriptions, the arguements refer to the words which make up the body of the message. The structure of a message to the process manager is:

```
struct {
        struct msghdr hdr;
        int pm_arg[];
}
```

Error codes are always returned in msustat. The possible errors are:

MSTYPERR	1	Message type is not recognized by the process manager
MSNOPRC	2	Process file does not exist or does not have the correct magic number.
MSNOFILE	3	A public library file does not exist.
MSNOSLOT	4	All the process slots in the system are in use
MSNOMEM	5	A kernel process could not be loaded because of insufficient memory
MSNOSWAP	6	All the system segments and/or swap space is allocated
MSREADERR	7	Process manager is unable to read the process file (the sender does not have read permissions on the process file)
MSNOINTR	10	A kernel process has attempted to connect to a nonexistent interrupt or an interrupt which is attached to another process.
MSVERSIONERR	11	The version number of a public library or the system library does not agree with the version number of the process

For the sake of completeness, the P_INIT and P_WAIT messages will be discussed. They are internal messages which are necessary for the process manager to maintain its internal tables.

NAME

P_INIT - initialize the process manager

SYNOPSIS

P_INIT = 4

INPUT

```
struct {
    struct msghdr hdr;
    int     maxprc;     /*Maxixmum number of processes in
                        system */
    int     syslib[2];  /*Creation date of system library */
    int     slsid;      /* Segment ID of kernel */
    int     sloff;      /* Offset into kernel segment of sys-
                        tem library */
    int     nprc;       /* Number of processes in boot image
                        */
    int     switch;     /* Console switch register */
    int     swapdev;    /* Major and minor device of swap
                        device */
    struct {
        char    pn;     /* Process number of boot process*/
        char    nseg;   /* Number of segments in the process
                        */
        char    iprior; /*processor priority of the process
                        */
        char    sprior; /* Scheduler priority */
        int     seg[3]; /*Segment IDs of the segments making
                        up the process */
    } blk[];
};
```

VALUES

none

DESCRIPTION

The init message is used to pass several system generation parameters to the process manager. The process manager initializes its internal tables, copies the PCB of the nub process into an internal buffer, then terminates the nub process.

DIAGNOSTICS

none

NAME

P_WAIT - Message type received when a process created by the process manager terminates

SYNOPSIS

```
P_WAIT = 5
```

INPUT

```
struct {
   struct msghdr hdr;
   int    buf[];
   };
```

VALUES

none

DESCRIPTION

The process manager updates its internal process tables and forwards the message to the parent of the terminating process.

DIAGNOSTICS

none

NAME

terminate - terminate a process and produce a core dump

SYNOPSIS

terminate = 3

INPUT

```
struct {
    struct msghdr hdr;
    char   uid;
    char   gid;
    int    pn;        /* process number to dump */
    int    segid;     /* Segment id of process PCB  (supervi-
                         sor only)*/
    int    pathindex; /*Offset to start of pathname */
    int    buf[];
};
```

VALUES

none

DESCRIPTION

If the process specified by pn is a kernel process, the segments are unlocked and returned to the system. If pathindex is nonzero, a core dump will be produced. Pathindex is the index into buf to the first character of the null terminated pathname for the core dump file. If pathindex = -1, the pathname will be the last part of the process file name appended to /cdmp (i.e. if the process file pathname was /dev/cd6, the core file would be /cdmp/cd6). If the dump is produced by a bpt or a bad kernel emt, the array buf will contain:

```
buf[0-5] - r0 - r5
buf[6]   - Reason for dump
buf[7]   - pc
buf[8]   - ps
```

The process manager will create a file having the same format as pfile produced by ldp, with the exception that the registers, code, pc, and ps will be placed in the last 9 words of the header block.

ALSO SEE

ldp(e), pfile(g)

DIAGNOSTICS

If the file pointed to by pathindex cannot be created, no dump will be produced and no error will be returned.

NAME

pcreat - create a process from a file

SYNOPSIS

pcreat = 1

INPUT

```
struct {
    struct msghdr hdr;
    int     cdir;      /*file descriptor of current directory
                       of sender*/
    char    uid;       /*User id */
    char    gid;       /*Group id */
    int     pn;        /*Process number of new process (re-
                       turned)*/
    int     parent;    /*Process number of parent*/
    char    flag;      /*If non-zero a message of type mtype
                       will be sent to parent when the process
                       dies*/
    char    mtype;      /*A message of mtype will be sent to
                       parent if flag is nonzero*/
    char    chan;      /*Control channel number of new pro-
                       cess*/
    char    arg;       /*Flag set if parent sends message to
                       new process*/
    int     des;       /* unused */
    int     share[2]; /* Segment to share with created
                       process(flags,ID)*/
    char    path[];    /Pathname of file to make into a pro-
                       cess*/
};
```

VALUES (returned)

pn - the process number of the new process

DESCRIPTION

Pcreat causes the process file specified by path to be loaded and executed. If the new process is a supervisor process, the process manager creates a mini-supervisor (called the nub) which reads the process file into the appropriate segments, and transfers control to it. If the new process is a kernel process, the process manager will create the appropriate segments and issue messages to the memory manager to process lock them in memory.

ALSO SEE

ldp(e), pcreat(f), pfile(g).

DIAGNOSTICS

Many error codes are returned. See the introduction to the process manager.

INTRODUCTION

The memory manager is part of the basic kernel. It executes at processor priority 2 with kernel D-space enabled. The primary function of the memory manager is to load the next process for the system scheduler. Since the memory manager contains all the code for managing physical memory, certain other services, primarily for the benefit of the process manager, are included.

NAME

load - load a process

SYNOPSIS

load = Ø

INPUT

```
struct {
    struct msghdr hdr;
    int    segid;      /*Segment ID of process PCB */
};
```

VALUES

user status = the number of segments read in to satisfy the request

DESCRIPTION

The PCB segment specified by segid is loaded, then all segments with the "pcbnn" or "pcbnxt" bits are loaded. Other segments which are not locked may be shifted in memory. Segments which are not nonswap or active may be swapped out.

DIAGNOSTICS

The user status is set to -1 if there is insufficient memory.

NAME

lock - process lock a segment

SYNOPSIS

INPUT

```
struct {
    struct msghdr hdr;
    int    segid;      /* Segment ID to lock */
};
```

VALUES

segid = physical address of the start of the segment

DESCRIPTION

The segment segid is loaded in memory contiguous with other process locked or nonswap segments and locked in memory.

DIAGNOSTICS
User status is set to -1 if there is insufficient memory.

NAME
term - terminate a process

SYNOPSIS
term = 3

INPUT

```
struct {
    struct msghdr hdr;
    char   uid;
    char   gid;
    int    pn;        /* process number to terminate */
    int    segid;     /* Segment id of process PCB */
    int    buf[];
};
```

VALUE
The same message is forwarded to the parent of the terminating process

DESCRIPTION
The process PCB segment segid is brought into memory. The user count on all the segments in the PCB is decremented and if zero, the segment is returned to the system.

DIAGNOSTICS
none

INTRODUCTION

The system scheduler is part of the basic kernel. It executes at processor priority 2 with kernel D-space enabled. The primary function of the scheduler is to schedule processes which execute at processor priority one and zero so as to maximize CPU usage without compromising real time response. The scheduler is primarily a message source, with almost all messages going to the memory manager, however, process termination requires manipulation of scheduler queues, hence the requirement that all terminate messages be sent to the scheduler.

NAME

term - terminate a process

SYNOPSIS

```
term = 3
```

INPUT

```
struct {
    struct msghdr hdr;
    char   uid;
    char   gid;
    int    pn;       /* process number to dump */
    int    segid;    /* Segment id of process PCB  (supervi-
                        sor only)*/
    int    pathindex; /*Offset to start of pathname */
    int    buf[];
};
```

VALUE

The same message is forwarded to either the memory manager or the process manager.

DESCRIPTION

The user count on the process pn is decremented and if zero, it is removed from the queue of active processes. All outstanding messages to the terminating process are returned with the system status byte equal to 0200. If the process is a kernel process or if pathindex is nonzero, the message is forwarded to the process manager, otherwise it is forwarded to the memory manager.

DIAGNOSTICS

The user status byte is set to -1 if pn is invalid.

NAME

check - file system consistency check

SYNOPSIS

check [-lsib [numbers]] [filesystem]

DESCRIPTION

Check examines a file system, builds a bit map of used blocks, and compares this bit map against the free list maintained on the file system. It also reads directories and compares the link-count in each i-node with the number of directory entries by which it is referenced. If the file system is not specified, a check of a default file system is performed. The normal output of check includes a report of

The number of blocks missing; i.e. not in any file nor in the free list,
The number of special files,
The total number of files,
The number of directories,
The number of blocks used in files,
The highest-numbered block appearing in a file,
The number of free blocks.

The -l flag causes check to produce as part of its output report a list of the all the path names of files on the file system. The list is in i-number order; the first name for each file gives the i-number while subsequent names (i.e. links) have the i-number suppressed. The entries ``.'' and ``..'' for directories are also suppressed.

The -s flag causes check to ignore the actual free list and reconstruct a new one by rewriting the super-block and bit map of the file system. The file system should be dismounted while this is done; if this is not possible (for example if the root file system has to be salvaged) care should be taken that the system is quiescent and that it is rebooted immediately afterwards so that the old, bad in-core copy of the super-block will not continue to be used. Notice also that the words in the super-block which indicate the size of the free list and of the i-list are believed. If the super-block has been curdled these words will have to be patched. The -s flag causes the normal output reports to be suppressed.

The occurrence of i n times in a flag argument -ii...i causes check to store away the next n arguments which are taken to be i-numbers. When any of these i-numbers is encountered in a directory a diagnostic is produced, as described below, which indicates among other things the entry name.

Likewise, n appearances of b in a flag like -bb...b cause the next n arguments to be taken as block numbers which are remembered; whenever any of the named blocks turns up in a file, a diagnostic is produced.

FILES

Currently, /dev/tf0, /dev/tf3 and /dev/tf1 are the default file systems.

SEE ALSO

fs (d)

DIAGNOSTICS

There are some self-evident diagnostics like ``can't open ...'', ``can't write ....'' If a read error is encountered, the block number of the bad block is printed and check exits. ``Bad freeblock'' means that a block number outside the available space was encountered in the free list. ``n dups in free'' means that n blocks were found in the free list which duplicate blocks either in some file or in the earlier part of the free list.

An important class of diagnostics is produced by a routine which is called for each block which is encountered in an i-node corresponding to an ordinary file or directory. These have the form

b# complaint ; i= i# (class)

Here b# is the block number being considered; complaint is the diagnostic itself. It may be

- blk if the block number was mentioned as an argument after -b;
- bad if the block number has a value not inside the allocatable space on the device, as indicated by the devices's super-block;
- dup if the block number has already been seen in a file;
- din if the block is a member of a directory, and if an entry is found therein whose i-number is outside the range of the i-list on the device, as indicated by the i-list size specified by the super-block. Unfortunately this diagnostic does not indicate the offending entry name, but since the i-number of the directory itself is given (see below) the problem can be tracked down.

The i# in the form above is the i-number in which the named block was found. The class is an indicator of what type of block was involved in the difficulty:

- sdir indicates that the block is a data block in a file;
- free indicates that the block was mentioned after -b and is free;
- urk indicates a malfunction in check.

When an i-number specified after -i is encountered while reading a directory, a report in the form

ino; i= d# (class) name

where i# is the requested i-number. d# is the i-number of

the directory, class is the class of the directory block as discussed above (virtually always ``sdir'') and name is the entry name. This diagnostic gives enough information to find a full path name for an i-number without using the -l option: use -b n to find an entry name and the i-number of the directory containing the reference to n, then recursively use -b on the i-number of the directory to find its name.

Another important class of file system diseases indicated by check is files for which the number of directory entries does not agree with the link-count field of the i-node. The diagnostic is hard to interpret. It has the form

i# delta

Here i# is the i-number affected. Delta is an octal number accumulated in a byte, and thus can have the value 0 through 377(8). The easiest way (short of rewriting the routine) of explaining the significance of delta is to describe how it is computed.

If the associated i-node is allocated (that is, has the allocated bit on) add 100 to delta. If its link-count is non-zero, add another 100 plus the link-count. Each time a directory entry specifying the associated i-number is encountered, subtract 1 from delta. At the end, the i-number and delta are printed if delta is neither 0 nor 200. The first case indicates that the i-node was unallocated and no entries for it appear; the second that it was allocated and that the link-count and the number of directory entries agree.

Therefore (to explain the symptoms of the most common difficulties) delta = 377 (-1 in 8-bit, 2's complement octal) means that there is a directory entry for an unallocated i-node. This is somewhat serious and the entry should be found and removed forthwith. Delta = 201 usually means that a normal, allocated i-node has no directory entry. This difficulty is much less serious. Whatever blocks there are in the file are unavailable, but no further damage will occur if nothing is done. A iclr followed by a check -s will restore the lost space at leisure.

In general, values of delta equal to or somewhat above 0, 100, or 200 are relatively innocuous; just below these numbers there is danger of spreading infection.

BUGS

Unfortunately, check -l on file systems with more than 3000 or so files does not work because it runs out of core.

Since check is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.

It believes even preposterous super-blocks and consequently can get core images.

NAME

df - disk free

SYNOPSIS

df [filesystem]

DESCRIPTION

Df prints out the number of free blocks available on a file system. It also prints out (in parenthesis) the number of blocks which are free but not in the in-core free list. If the file system is unspecified, the free space on all of the normally mounted file systems is printed.

FILES

/dev/rf?, /dev/rk?, /dev/rp?

SEE ALSO

check(VIII)

BUGS

NAME

falloc - allocate contiguous file space

SYNOPSIS

falloc filename nblocks

DESCRIPTION

Falloc creates a new file specified by the given pathname filename with default mode bits of 0666. The contiguous mode bit is set for this file. Contiguous file space is allocated on secondary storage to make room for nblocks of 512 bytes each. The file is initially of length zero.

This command is useful for creating large files, to/from which the user wishes to guarantee large data transfers using physical I/O and minimum latency time.

DIAGNOSTICS

If the file already exists or the directory in which the file is to be created is not writeable, the file is not created. If insufficient file space exists, the largest possible contiguous file space will be allocated to this file.

SEE ALSO

falloc(c), fs(d).

BUGS

NAME

fmove - move file into contiguous area

SYNOPSIS

fmove [-] filename

DESCRIPTION

Fmove moves the file filename into a contiguous secondary storage area using only one extent to describe its allocation area on disk. If the optional argument "-" is not specified and the file cannot be moved to a lower area on disk than it currently is, it will not be moved. Specifying this optional flag over-rides this restriction and the file is moved anyways.

This command is useful for moving a file into a contiguous area, to/from which the user wishes to guarantee large data transfers using physical I/O and minimum latency time.

DIAGNOSTICS

SEE ALSO

falloc(c), fs(d), falloc(d).

BUGS

NAME

fs - format of file system volume

DESCRIPTION

Caution: this information applies only to the latest versions of the MERT file system.

Every file system storage volume (e.g. RF disk, RK disk, RP disk, DECtape reel) has a common format for certain vital information. Every such volume is divided into a certain number of 256 word (512 byte) blocks. Block 0 is unused and is available to contain a bootstrap program, pack label, or other information.

Block 1 is the super block. Starting from its first word, the format of a super-block is

```
struct {
        char    *s_isize;       /* number of blocks of inodes */
        char    *s_fsize;       /* number of blocks in file system
        struct  {
                char    *stblk;
                char    *ncblks;
        } s_ext[64];
        int     s_ninode;       /* number of free inodes */
        int     s_inode[60];
        int     s_nupdate;      /* number of update entries filled
        struct  {
                char    *stblk;
                int     nublks;
        } s_update[30];
        char    s_flock;
        char    s_ilock;
        char    s_fmod;
        char    s_ronly;
        int     s_time[2];
};
```

Isize is the number of blocks devoted to the i-list, which starts just after the super-block, in block 2. Fsize is the first block not potentially available for allocation to a file. This number is unused by the system, but is used by programs like check (d) to test for bad block numbers.

The free list for each volume is maintained as follows. The in-core free list for each mounted volume consists of 64 double-word entries. The first word in an entry is the first free block stblk of the number ncblks of consecutive free blocks described by this extent. The free list for each volume is also maintained in a bit map kept on the volume starting just beyond the blocks devoted to the i-list. A number of update entries s_nupdate are kept in the in-core list s_update to keep the bit map uptodate. These update entries consist of double word entries, a starting block number and number of consecutive blocks.

To allocate nb blocks, the in-core free list is searched for the best entry to use. The algorithm used is to search the list for the smallest entry from which nb blocks can be allocated. If there are not nb free blocks, the largest entry is chosen. If there are no free blocks, the in-core free list is reconstructed using the bit map maintained on the volume. If there are still no free blocks, an error is returned. The in-core free list is updated and an entry is put in the update list. When the update list becomes full, the bit map is updated on the volume using the in-core update list and the update list is marked empty.

To free nb blocks, the in-core free list is searched according to the following algorithm. The nb blocks are added to an existing entry if they are contiguous with it. The entry is put in a null entry if one exists. If there is no empty entry, the smallest entry is replaced by the new entry. This entry is also put on the update list with a negative block count to distinguish it from an "alloc" entry. When the update list becomes full, the bit map is updated on the volume using the in-core update list and the update list is marked empty.

Ninode is the number of free i-numbers in the inode array. To allocate an i-node: if ninode is greater than 0, decrement it and return inode[ninode]. If it was 0, read the i-list and place the numbers of all free inodes (up to 100) into the inode array, then try again. To free an i-node, provided ninode is less than 100, place its number into inode[ninode] and increment ninode. If ninode is already 100, don't bother to enter the freed i-node into any table. This list of i-nodes is only to speed up the allocation process; the information as to whether the inode is really free or not is maintained in the inode itself.

Flock and ilock are flags maintained in the core copy of the file system while it is mounted and their values on disk are immaterial. The value of fmod on disk is likewise immaterial; it is used as a flag to indicate that the super-block has changed and should be copied to the disk during the next periodic update of file system information. Ronly is a flag used to indicate that the file system is read-only, i.e. no files may be modified.

Time is the last time the super-block of the file system was changed, and is a double-precision representation of the number of seconds that have elapsed since 0000 Jan. 1 1970 (GMT). During a reboot, the time of the super-block for the root file system is used to set the system's idea of the time.

I-numbers begin at 1, and the storage for i-nodes begins in block 2. Also, i-nodes are 64 words long, so 4 of them fit into a block. Therefore, i-node i is located in block (i + 7) / 4, and begins 1288.9((i + 7) (mod 4) bytes from its start. I-node 1 is reserved for the root directory of the file system, but no other i-number has a built-in meaning.

Each i-node represents one file. The format of an i-node is as follows.

```
struct {
        int     flags;  /* +0: see below */
        char    nlinks; /* +2: number of links to file */
        char    uid;    /* +3: user ID of owner */
        char    gid;    /* +4: group ID of owner */
        char    fill;   /* +5: used internally */
        int     size0;  /* +6: high byte of 32-bit size */
        int     size1;  /* +8: low word of 32-bit size */
        struct {
                int *stblk;     /* starting block number */
                int *ncblks;    /* number of cons. blocks */
        } extents[27];
        int     actime[2];      /* +118: time of last access */
        int     modtime[2];     /* +122: time of last modification
        int     chksum; /* not used */
};
```

The flags are as follows:

```
 100000   i-node is allocated
 070000   3-bit file type:
      000000   plain file
      040000   directory
      020000   character-type special file
      060000   block-type special file
      070000 record-type special file.
 010000   contiguous file
 004000   set user-ID on execution
 002000   set group-ID on execution
 000400   read (owner)
 000200   write (owner)
 000100   execute (owner)
 000070   read, write, execute (group)
 000007   read, write, execute (others)
```

Special files are recognized by their flags and not by i-number. A block-type special file is basically one which can potentially be mounted as a file system; a character-type special file cannot, though it is not necessarily character-oriented. For special files the high byte of the first extent word specifies the type of device; the low byte specifies one of several devices of that type. The device type numbers of block and character special files overlap.

The extent words of ordinary files and directories contain pairs of starting block numbers and number of consecutive blocks. As many extents are used as are required to describe the discontinuous pieces of the file. A contiguous file requires only one extent.

Byte number n of a file is accessed as follows. N is divided by 512 to find its logical block number (say b) in the file. The physical block number is the b th logical block in the list of extents. The remainder from the division

yields the byte in the block which is to be accessed.

SEE ALSO
check (d)

NAME

icat - concatenate i-node

SYNOPSIS

icat [filesystem] i-number ... i-number

DESCRIPTION

Icat concatenates the contents of the file described by the i-node numbered i-number. An arbitrary number of i-node numbers may be specified. The file system argument must be a special file name referring to a device containing a file system.

Read permission is required on the specified file system device. The primary purpose of this routine is to concatenate the contents of a file which has an i-node allocated but no corresponding entry in any directory. By using "icat [filesystem] i-number > savefile", it is possible to save the contents of a file on another filesystem before using "iclr" to clear the i-node entry.

SEE ALSO

iclr(d), idmp(d), isnp(d).

BUGS

NAME

iclr - clear i-node

SYNOPSIS

iclr [filesystem] i-number ... i-number

DESCRIPTION

Iclr writes zeros on the 64 words occupied by the i-node numbered i-number. An arbitrary number of i-node numbers may be specified. The file system argument must be a special file name referring to a device containing a file system. After iclr, any blocks in the affected file will show up as ``missing'' in a check of the file system.

Read and write permission is required on the specified file system device. The i-node becomes allocatable.

The primary purpose of this routine is to remove a file which for some reason appears in no directory. If it is used to zap an i-node which does appear in a directory, care should be taken to track down the entry and remove it. Otherwise, when the i-node is reallocated to some new file, the old entry will still point to that file. At that point removing the old entry will destroy the new file. The new entry will again point to an unallocated i-node, so the whole cycle is likely to be repeated again and again.

SEE ALSO

icat(d), idmp(d), isnp(d).

BUGS

If the file is open, iclr is likely to be ineffective.

NAME

idmp - dump i-node

SYNOPSIS

idmp [filesystem] i-number ... i-number

DESCRIPTION

Idmp dumps the contents of the file described by the i-node numbered i-number. An arbitrary number of i-node numbers may be specified. The file system argument must be a special file name referring to a device containing a file system.

Read permission is required on the specified file system device. The primary purpose of this routine is to dump the contents of a file which has an i-node allocated but no corresponding entry in any directory.

SEE ALSO

iclr(d), icat(d), isnp(d).

BUGS

NAME

isnp - snap i-node contents

SYNOPSIS

isnp [filesystem] i-number ... i-number

DESCRIPTION

Isnp snaps the contents of the i-node numbered i-number. An arbitrary number of i-node numbers may be specified. The file system argument must be a special file name referring to a device containing a file system.

Read permission is required on the specified file system device. The primary purpose of this routine is to snap the contents of the i-node.

SEE ALSO

iclr(d), icat(d), idmp(d).

BUGS

NAME

mknod - build special file

SYNOPSIS

/etc/mknod name [c] [b] major minor

DESCRIPTION

Mknod makes a directory entry and corresponding i-node for a special file. The first argument is the name of the entry. The second is b if the special file is block-type (disks, dectape), r if it is record-type (magtape, TIU) or c if it is character-type (other devices). The last two arguments are numbers specifying the major device type and the minor device (e.g. unit, drive, or line number).

The assignment of major device numbers is specific to each system. The major device number corresponds to the appropriate device driver process in "/prc" directory.

SEE ALSO

mknod (II)

BUGS

NAME

mkpt - make prototype file for use by mkfs

SYNOPSIS

/etc/mkpt spec proto

DESCRIPTION

Mkpt constructs a prototype file proto according to the directions found in the specification file spec. The specification file must contain the name of the boot file as the first token, the size of the file system to be created as the second token and the size of the i-list in blocks as the third token. The next set of tokens comprise the specification for the root file. See mkfs (VIII) for the correct format of the above tokens.

All of the following tokens consist of directory names. Directory names followed by a ':' are made directory entries in the proto file. Directory names not terminated by a ':' are scanned by mkpt and all entries are put in the proto file (recursively). If no ':' terminated directory name appears, all entries taken from the specified directories are entered into the proto file as originating from the root directory. A directory name terminated by a ':' and followed immediately by another directory name terminated by a ':' will be entered as an empty directory entry in the proto file. A directory name terminated by a ':' may be followed by as many directory names not terminated by ':'s as desired.

A directory name terminated by a ':' may be several levels deep, e.g. usr/usra: . All subsequent directory names terminated by ':' will be put in the directory 'usr' . One may back up a directory level by means of the entry '..:' . This enables one to extract selectively certain subdirectories of a given directory.

Editing of the proto file will only be required if one wisnes to change the names of some files. Otherwise the proto file may be used exactly as is to create a new file system using mkfs (see mkfs (VIII)).

Some sample specification files follow:

```
/usr/mdec/uboot
4872 55
d--777 3 1
bin: /bin
lib: /lib
dev: /dev
etc: /etc
tmp:
usr:
mnt:
crp:
```

and

```
/usr/mdec/uboot
42000 500
d--777 3 1
/usr
```

and

```
/usr/mdec/uboot
20000 300
d--777 3 1
usra: /usr/usr1 /usr/usr2 /usr/usr3
usrb: /usr/usr4
usrc:
usrd: /usr/usr5
```

and

```
/usr/mdec/uboot
35000 450
d--777 3 1
/
```

and

```
/usr/mdec/uboot
4000 80
d--777 3 1
bin: /bin
etc: /etc
dev: /dev
lib: /lib
tmp:
usr/usra: /usr/usra
usrb: /usr/usrb
usrc:
bin: /usr/bin
mnt:
crp:
```

SEE ALSO

file system(V), directory(V), boot procedures(VIII).

DIAGNOSTICS

There are diagnostics for non-existent or non-readable directories.

BUGS

If a very large proto file is created, it cannot be edited to make any changes.

NAME

recdmn - reconfiguration daemon

SYNOPSIS

/etc/recdmn

DESCRIPTION

Recdmn is the file system reconfiguration daemon. It checks every 30 minutes to see if the time is between 3AM and 4AM and then performs reconfiguration on the file systems which are specified in its internal table only if it can successfully unmount the file system. Thus the file system must be inactive for a reconfiguration to be performed. The internal table consists of entries with pointers to 3 ascii strings, e.g.:

"/dev/tf3", "/usr", "-s",

Here the first string specifies the file system to be unmounted, the second string specifies the name of the directory under which the file system is to be mounted upon completion of the reconfiguration, and the third string specifies the algorithm to be used in the compaction of the file system.

The diagnostics produced by the daemon are written on the file "/etc/recdlog" and may be examined the next morning for a post-mortem. Each night that the reconfiguration daemon runs, it starts writing at the beginning of the diagnostic file again.

FILES

/dev/rf?, /dev/rk?, /dev/rp?, /etc/recdlog

SEE ALSO

recon(d), check(d).

BUGS

NAME

recon - reconfigure file system

SYNOPSIS

recon [-x] [r] [highwater] filesystem

DESCRIPTION

Recon examines a file system, builds a bit map of used blocks from the contents of the i-nodes, and reconfigures files on the file system so as to compact all files onto the lower blocks of the file system, making the files contiguous if possible. Three basic algorithms may be used to compact a file system. The first optional flag is used to specify which of the algorithms are to be used. The -a option causes all files to be compacted and relocated to lower block numbers only if enough contiguous file space can be found. The search starts with i-node 1 and goes up to the highest i-node in the file system. The search may be done in reverse order starting with the highest i-node number in the file system by specifying the second r flag. Files which are contiguous, i.e. marked contiguous by bit in i-node flag, are not moved under any circumstances.

The -b option causes only files above a certain highwater block number to be searched as candidates to be relocated to lower block numbers. After the search for all possible candidates, the i-nodes are sorted in order of file with the highest block number first. Thus the files with higher block numbers will be relocated to lower holes before moving those files which are already at lower block numbers initially.

The -s option causes only files above a certain highwater block number to be searched as candidates to be relocated to lower block numbers. After the search for all possible candidates, the i-nodes are sorted in order of largest file first. Thus the largest file will be moved first to fill in any existing holes in the bit map. The smaller files will fill in the holes that are left over.

In both -b and -s options the r flag may also be used to do the sorting in the reverse order. Also if the argument after the flags is numerical, it will be taken as the actual highwater block highwater in the search for files to be moved. The -s option is taken as a default option.

The filesystem must be unmounted before a reconfiguration is attempted on it. Once recon has started to move files, all signals are ignored so that the program cannot be aborted when partially completed.

FILES

There is no default file system.

SEE ALSO

fs(d), check(d), recdmn(d).

DIAGNOSTICS

If a read error is encountered, the block number of the bad block is printed and recon exits.

BUGS

There is currently no check against applying recon to an active file system. It believes even preposterous superblocks and consequently can get core images.

NAME

acp - asynchronous copy

SYNOPSIS

acp file1 file2

DESCRIPTION

The first file is copied onto the second file using asynchronous I/O directly from/to user's address space. Reads and writes are done in 3K word blocks. One read and one write are outstanding simultaneously. The mode and owner of the target file are preserved if it already existed; the mode of the source file is used otherwise.

If file2 is a directory, then the target file is a file in that directory with the file-name of file1.

SEE ALSO

cat(I), pr(I), mv(I), cp(I), pcp(e).

BUGS

Copying a file onto itself destroys its contents.

NAME

kdb - kernel debugger

SYNOPSIS

kdb [-] [namelist]

or

kdb [core [namelist]]

DESCRIPTION

Kdb is a debugging package for the kernel layer of the MERT operating system. Like the UNIX debugger, kdb is used to examine core image files. Typically, the file will be either a core image of a system resulting from a crash or /dev/mem when examining the current operating system.
The command line has two forms:

If a minus (-) is specified, kdb will use /dev/mem as the core file. References to segments which are not currently in memory will be satisfied by reading the segment from /dev/swap.

If a core file is specified, kdb will treat all references to the segments which are not in the core image as errors.

If no arguments are given, the default core file is kore in the current directory and the default namelist is /mrt/krn.sym.

The format of kdb requests is a one or two character mneumonic followed by a list of parameters. Numeric parameters are assumed to be octal unless terminated by a decimal point. In the following description only the first two characters of mneumonics are required:

$

The user, supervisor, and kernel stack pointers, followed by the general registers are typed. This command causes kdb to set its internal tables for virtual address mapping of the kernel address space to that at the time the kore image was produced. This should not be used when debugging /dev/mem.

! command line

The exclamation point (!) is stripped and the rest of the line is sent to the shell for execution.

*

The values of the kernel segmentation registers (sdr and sar) are displayed.

= symbol

The value of the symbol symbol is displayed.

dct pn1 pn2

The Dispatcher Control Tables starting at process pn1 through pn2 are displayed. If pn1 is not entered, all the DCTs are typed.

dl pri1 pri2 0 < pri1 <= 7 pri1 <= pri2 <= 7

The Dispatcher Control Tables (DCT) entries which are on the processor priority chain pri1 are list-

ed in the order inwhich they appear on the linked list. After all DCT entries are listed, pri1 is incremented. If pri1 is less than pri2, the next chain is listed.

ml

The Segment Discriptor Entries (SDE) are listed in order of increasing memory.

msg

The contents of all the message buffers are displayed.

pmsg pn1 pn2

The contents of all messages on the message queues of processes pn1 through pn2 are displayed.

rsde segid

The RSDE (Resident Segment Descriptor Entry) for the segment segid is displayed. If segid is not specified, all RSDEs are displayed.

sde address

The SDE (Segment Descriptor Entry) pointed to by address is displayed. If no address is specified all SDEs are displayed.

seg segid offset n

The contents of the segment segid starting at byte offset offset into the segment and continuing for n bytes is displayed. If n is not specified, the end of the segment is assumed. If offset is not specified, zero is assumed.

scan start end pattern mask

Kernel virtual address space is searched starting at start up to and including end for a match on the pattern pattern. Each word of kernel memory is masked with mask before the comparisons are made.

snap addr[d] n

The kernel virtual address addr and n consecutive bytes are typed. If the address is followed by a "d" (no blanks or tabs) the address is interpreted as a D-space address, otherwise I-space is assumed. Virtual address to file offset is done via the image of the segmentation registers in kdb's internal tables. If /dev/mem is being examined, there is no way to get the current setting of the segmentation registers. If a core file is being examined, the '$' command will set kdb's mapping tables to those of the system at the time the core image was produced.

xt pn

Extracts an image of the process pn into the file korexxx, where xxx is the octal index into the DCT tables for the process. The program sdb can be used to examine these core images.

ALSO SEE
sdb(e),kdmp(e),tdmp(e)

DIAGNOSTICS
"Open err:" if a file cannot be opened, otherwise "?".

FILES
kore core image file
/dev/mem
/dev/swap
/mrt/krn.sym namelist

NAME

kdmp - dump system state into core file

SYNOPSIS

kdmp [korefile] [diskfile]

DESCRIPTION

Kdmp reads the state of the system as it was saved on disk when the system "panic'ed" producing a dump of all of memory starting on a fixed disk cylinder. If no korefile is specified, the dump is put in the file "kore" in the current directory. If no diskfile is specified, the dump is taken from the default "diskfile" compiled into kdmp.

The starting cylinder numbers of the system dump for the various disks are:

Telefile	200.
RP03	200.
RP04	408.
DECtape	0.

The "kore" file is suitable for debugging with kdb.

SEE ALSO

kdb(e), tdmp(e).

BUGS

NAME

ldp - load a process

SYNOPSIS

ldp file

DESCRIPTION

Ldp breaks programs up into segments, performs relocation of the text, data, and bss, satisfies external references to shared data segments (data libraries), and shared code segments (public libraries), and provides the machinery for defining symbols (of type common) in segments other than the process data and bss segment.

Input to ldp is a specification file which consists of lines containing a colon (:) terminated keyword and a parameter list. The elements in the parameter list are separated by blanks, tabs, commas, or if enclosed in braces ({}), by new lines. Numerical parameters are assumed to be octal unless terminated by a decimal point. In the discussion of the keywords parameters enclosed in brackets ([]) are optional, address means the ascii name of an external symbol, and ps is the octal value of the processor status word.

bss: symbol [symbol ...]

The bss option permits symbols of type common to be allocated at the beginning if the data segment rather than at the end. This option is useful if one is trying to insure that a set of structures begin on a 32 word boundary. The keyword bss is a misnomer since symbols of type bss can not be relocated using this option.

common: sr [-] symbol [symbol ...]

The common option defines a shared segment starting in segmentation register sr and assigns the symbols in the parameter list to the segment. Only symbols of type common (any uninitialized C structure) can be assigned. The size of the segment is equal to the sum of the storage required by the symbols in the parameter list. Up to eight common segments may be specified for supervisor or user processes and two for kernel processes. The common keyword must be repeated for each segment. The segment will contain all zeros when created. Subsequent invocations of the process (via a create message to the process manager) will simply increment the user count on the segment. Ldp allocates a dummy block of 512 bytes at the end of the process data for each common block. This is required in order to associate a unique name with the corresponding segment. If the minus (-) parameter is specified, the common segment will be set up as a stack segment (growing to lower addresses). The first symbol address will be computed according to the C statement:

```
addr = (sr<<13) + 8192 - sizeof(symbol);
```

Successive symbols are assigned lower addresses.

copy: n

This specifies the maximum number of invocations of the process. For a process such as the disk driver n is usually 1. For the unix supervisor, n is normally greater than the number of available processes in the system.

data: pathname [pathname ...]

References to symbols in shared data segments are satisfied. The data segments are placed in the holes in the process address space after all space has been allocated for public libraries, process text, process private data, and stack. Pathnames must begin at the root file system and must be the same in the system under which ldp is running as in the system the process will run.

database: sr [s]

The data and bss sections of the public library are loaded starting in segmentation register sr. The s parameter causes this segment to be shared by multiple invocations of the library. This is useful for generating system libraries where the text, data, and bss are combined in one sharable, writable segment. This keyword is only meaningful if a public library is being created (mode: p).

dcom:

The dcom option (for data common) causes the process data segment to be shared (normally the text segment is shared and a new copy of the data segment is made). The segment always appears as the third entry in the PCB segment table for supervisor processes; for kernel processes the data segment will begin at the next 8K byte boundary.

dsect: sr [-] symbol [symbol ...]

The dsect keyword is short hand for dummy section. It permits symbols of type common (eg uninitialized C structures) to be assigned values corresponding to segmentation register sr (eg if sr = 3, the first symbol would have the address 060000). By default, successive symbols are assigned higher addresses. Symbol addresses may cross segmentation register boundaries and rap around. If the minus (-) parameter is given, the address assigned to the first symbol is given by the C statement:

```
addr = (sr<<13) + 8192 -sizeof(symbol);
```

Successive symbols are allocated at lower addresses. No logical segment is created by the dsect option; it is simply a way of controlling the addresses of structures without knowing their sizes.

emt: address

Address specifies the entry point into a kernel process for servicing emt traps from supervisor processes.

entry: address [ps]
Address specifies the initial entry point into a supervisor process.

event: address [ps]
Address specifies the entry point into a supervisor process for receiving event interrupts.

fault: address [ps]
Address specifies the entry point to handle all traps (except emt and bpt from supervisor or kernel mode).

idchar: c
The character c is passed the process manager in the process file header. The process manager passes the character to the kernel process tables where it is available for identifying the process with the ps program. This option is only meaningful for kernel processes.

ifile: pathname [pathname] ... [-lx] ...
This option specifies the list of object files to be link-edited together to form the process image. The normal shell syntax may be used (eg *.o abc? x[r-z]*.o). The "-lx" is short hand for /lib/libx.a and is used to specify archive format libraries. The modules are loaded and libraries searched in the order specified in the parameter list. The system library (if a kernel process so requests), any public libraries, /lib/libe.a (for supervisor processes) or /lib/libk.a (for kernel processes), /lib/libs.a, and /lib/libc.a are then searched in order.

interrupt: device entry
or
vector entry
The parameter list specifies the vector addresses and the corresponding entry point for all interrupts the process expects to handle. The specification of the vector address can be by vector, the octal address of the interrupt vector, or device, a symbolic name for one of the standard DEC peripherals:

rconsole	keyboard part of console teletype
xconsole	printer part of console teletype
pc11r	paper tape reader
pc11p	paper tape punch
kw11p	programmable clock
parity	parity memory
pl	xyplotter
lp	lp11 line printer
ls	ls11 line printer
rf	rf11 disk
rc	rc11 disk
tc	tc11 DEC tape
rk	rk11 disk

tm	tm11 magnetic tape
cr	cr11 card reader
cd	cd11 card reader
cm	cm11 card reader
rp	rp03 disk
tf	telefile rp03 equivalent disk
ta	cassette tape

A process can specify up to 32 interrupts.

mode: ksup[l][d][i]

Specifies whether the output file is a public library (p), kernel (k), supervisor (s), or user (u) process. The l option causes the system library symbol table to be searched before the libraries /lib/libk.a and /lib/libs.a (kernel processes only). The d (or i) option causes data and bss to be relocated to the data segmenatation registers.

ofile: pathname

The process file is deposited in pathname.

open:

This keyword specifies that the process expects an open I/O message from the file system whenever a device handled by the process is opened and a close I/O message whenever a device is closed.

pcbbase: sr

The segmentation register allocated to the process PCB is sr. The default value is zero.

priority: n

For kernel processes n is the actual processor priority at which the processor (bits 7:5 of the ps) is set while the process is executing (3 <= n <= 7). For supervisor processes n is the scheduler priority of the process (0<= n <= 260).

publib: pathname [pathname ...]

The public libraries specified by the parameter list are searched during the binding phase of process building. The pathnames must start from the root file system and be the same in the system under which ldp is running as the system under which the process will run.

share: sr access [symbol symbol ...]

The share option permits a process to specify that a shared segment is to be provided by the creating process, and that the shared segment is to appear in the process virtual address space starting in segmentation register sr. The shared segment will be given access permission access (2 for read only, 6 for read/write), and the symbols (of type common) will be given addresses starting at the beginning of the segment. For supervisor processes, the shared segment will appear in the PCB segment table as the third entry (after the PCB, stack, and text) if the data and stack are combined (see stack option) or as the fourth entry (after the data segment) if the data and stack are not combined.

stack: size sr [d]

The stack segment of size bytes will be allocated

starting in segmentation register sr and occupying successive (lower numbered) segmentation registers. The d option causes data, bss, and common symbols to be loaded at the top (high address end) of the stack segment. If the d option is specified, the size of the stack will be:

data size + bss size + common size + size.

The stack pointer will be initialized to point to the appropriate virtual address:

sp = (sr <<13) + 8192 - data size - bss size;

or

sp = (sr <<13) + 8192;

The stack segment always appears as the second entry in the PCB segment table.

swap: start nblks

The swap keyword declares the process as a valid candidate for supporting system swapping. Start is the block number (relative to the start of the logical device defined by sgen (e)), and nblks is the number of blocks in the swap area. An example for an rk disk might be:

swap: 4000., 872.

Note that there are no checks on logical device between sgen and ldp, hence an incorrect specification of the logical device could result in a file system being destroyed.

time: n

The length of the process time slice (in 1/60ths second) is set to n.

textbase: sr

The starting segmentation register for the text sharable segment of a public library is set to sr. If the database keyword is not specified, all data and bss will be included in the text segment and thus will be read only. The textbase keyword is only meaningful when a public library is being built (mode: p).

The specification file for the unix supervisor is:

```
mode:       s
pcbbase:    5
dcom:
sp:         1024.,7
dsect:      7,_u
ifile:      *.o
ofile:      /etc/unix
entry:      start,030000
fault:      trap,030000
event:      event,030040
time:       120.
```

The specification file for the kernel process which handles the dhll device is:

```
mode:       kl
priority: 5
interrupt:{
            320,_dhrint
            324,dhxint
```

```
                314,_dmint
        }
        emt:    dhemt
        event:  dhevent
        ifile:  dh.o dhdm.o dhmch.o
        ofile:  /dev/cdevl
```

FILES

/bin/ld
/bin/sh
a.out

ALSO SEE

ps(1)

DIAGNOSTICS

Error conditions are classified as being either fatal or non-fatal. The following warning messages are output on the occurrence of a non-fatal error

Bad option	A bad keyword in the specification file - the line is ignored.
Event entry point ??	No entry point was specified for events.
Intr. vectors ??	No interrupts were specified for a kernel process.
U: symbol	The symbol symbol is undefined

NAME

ldu - load a user process with public libraries

SYNOPSIS

ldu [-in] name.o ... [-p plib] [-o ofile]

DESCRIPTION

Ldu provides an aid to forming a UNIX user program which uses public libraries. It forms a temporary file with process specifications which are then given to the ldp program (ldp(e)). The default parameters are:

```
mode:   uc
ifile:  /lib/crt0.o name.o
ofile:  p.out
stack:  02400.
```

The optional keys which may be specified are:

-i program text and data areas are separated into I and D space.

-n text portion of output file will be read-only and shared by all users executing the output file.

-p the name of the file following this option will be taken as a public library. The symbol names in the public library file are linked into the main program. The public library is loaded when the executable output file is loaded.

-o the name of the output file may be specified by the name following the option key.

In the default case, the main program has text and data combined. Up to 3 public libraries may be included in the output file.

SEE ALSO

ldp(e), ll(e).

BUGS

NAME

ll - library loader

SYNOPSIS

ll -dxsxtx

DESCRIPTION

LL takes a file in a.out format, with relocation bits and no undefined external symbols, relocates it and inverts the symbol table and program. The resulting file (called a public library) can be used by the loader to satisfy references to symbols defined in the file without including the code in the loader output file. The flags define the type of relocation to be performed, relocation bias, and access to be associated with the library.

- d - Use D-space, the number x is the segmentation register in which the data is to reside.
- s - The library text segment is to execute in segmentation register x. The text is given read/write permissions. If the d option is not specified, the data will be appended to the text. This option is used to construct the system library.
- t - The library text is to execute in segmentation register x. The text is given read only access. If the d option is not specified, the data will be appended to the text and will have read only access.

FILES

a.out

ALSO SEE

ldp (e), sgen (e)

DIAGNOSTICS

?? A letter other than d, s, or t is given as a flag.

Bad base register x does not satisfy 0 <= x <= 7.

NAME

pcp - physical copy

SYNOPSIS

pcp file1 file2

DESCRIPTION

The first file is copied onto the second file using physical I/O directly from/to user's address space. Reads and writes are done in 3K word blocks. The mode and owner of the target file are preserved if it already existed; the mode of the source file is used otherwise.

If file2 is a directory, then the target file is a file in that directory with the file-name of file1.

SEE ALSO

cat(I), pr(I), mv(I), cp(I), acp(e).

BUGS

Copying a file onto itself destroys its contents.

NAME

pio - physical I/O

SYNOPSIS

pio command

DESCRIPTION

The given command is executed using physical I/O to/from all files where possible. If physical I/O is not possible, the system does the appropriate side-buffering.

This command is useful for doing "check", "dd" and other UNIX type commands which make use of raw I/O.

SEE ALSO

cp(I), pcp(e), acp(e).

BUGS

NAME

ps - process status

SYNOPSIS

ps [aklstx] namelist

DESCRIPTION

Ps prints certain indicia about active processes. The a flag asks for information about all processes with teletypes (ordinarily only one's own processes are displayed); t asks for all processes with the teletypes named in next argument. x asks even about processes with no typewriter; l asks for a long listing; k asks for a listing of all kernel processes; s permits the specification of an alternate swap device. The pathname of the swap device must be the next argument. Namelist will be used for the kernel symbol table instead of /mrt/krn.sym. Ordinarily only the typewriter number (if not one's own) and the process number are given.

The long listing is columnar and contains

A word containing the process status flags.

The priority of the process; high numbers mean high priority.

The amount of kernel, supervisor, and user time consumed by the process (in 1/60th second) is displayed in the next three words.

The last character of the control typewriter of the process.

The process unique number (as in certain cults it is possible to kill a process if you know its true name).

The size (in 32 word blocks) of the process (includes supervisor segments).

The bit pattern on which the process is sleeping.

A guess at the command line used to envoke the process.

The list of attributes for the kernel processes contains:

The process status flags.

The hardware priority at which the process executes.

The head of the process message queue or zero if the queue is empty.

The number of clock ticks until the next time out event or blank if the process is not timing.

The process event flags word.

The process control channel number (always Ø376).

The process unique number.

The physical (block) memory address of the start of the process.

The octal index into the process tables (DCT tables) for the process.

A one character identifier which is specified at process build time.

The name of the controller or device which the process handles

Plain ps will tell you only a list of numbers.

FILES

/mrt/krn.sym	system namelist
/dev/swap	swap device
/dev/mem	resident system
/mrt/kprc	names of controllers for kernel processes

SEE ALSO

kill(I) tkill(e) ldp(e)

NAME

run - run an environment

SYNOPSIS

run [-b] [-f] file

DESCRIPTION

A new environment (task) is started up. The specifications of the task are contained in the file file. The file is built using ldp (see ldp(e)).

A new process is created from the contents of file. The death of this process is waited for only if the b option is not specified to run the process in the background. A non-zero flag is passed to the new process if the f option is specified.

SEE ALSO

pcreat(f), wait(II).

BUGS

NAME

sgen - system generation program

SYNOPSIS

sgen [-uv] spec [file]

DESCRIPTION

Sgen is a program which builds a core image of the basic MERT operating system. The functions performed are:

1) It generates the low core image, allocating interrupt vectors and establishing the linkage by which processes can attach to the interrupts at run time.
2) Relocates the kernel text and data segments so that they each start at 20000(8).
3) Appends the basic modules needed at boot to the kernel core image. These include:
    1) The system library
    2) The process for the root device
    3) The process for the swap device
    4) The file manager process
    5) The process manager bootstrap process
    6) The nub process (a subtask of the process manager)
    7) System initialization process
4) Generates a table of pathnames of processes to be created by the process manager at boot time.

Input to sgen consists of flags and a specification file. The flags are:

u Do an update sysgen, that is do not regenerate a new low core image.

v Verbose mode - print out a map of memory at the end of system generation.

The specification file consists of lines containing a keyword and a parameter list. The elements in the parameter list may be separated by blanks, tabs, or commas. A comment delimited by /* may be added to any line. All numerical parameters are assumed to be octal unless terminated by a decimal point. The keywords are:

fmgr pathname
The file pathname contains the process file (the output of ldp) of the file manager process

init pathname
The file pathname contains the relocatable initialization process.

kernel pathname
The file pathname contains the relocatable kernel.

lowcore pathname
The file pathname contains the relocatable binary of the lowcore module. This module is generated by sgen from the assembly language files lcor0.s and lcor1.s.

nubprc pathname
The file pathname contains the process file (output of ldp) of the process which creates supervisor mode processes. This process is actually a subtask of the process manager. It is included in system generation to simplify booting.

pmboot pathname
The file pathname contains the process file (the output of ldp) of the process which will create the process manager.

pmgr pathname
The file pathname contains the process file of the process manager. Sgen simply passes this pathname to the init process which in turn passes it to pmboot. The pathname must start from the root and the file must exist on the root file system at boot time.

rootdev major minor
Major is the major device number of the root file system. Minor is the minor device number of the root file system.

rootprc pathname
The file pathname is the process file of the process which services the device containing the root file system.

swapdev major minor
Major is the major device number of the swap device. Minor is the minor device number of the swap device.

swapprc pathname
The file pathname is the process file of the process which services the device which contains the system swap area.

syslib pathname
The file pathname contains the public library file (output of ll) set up to execute in kernel base register six. If this keyword is excluded, no system library will be generated. By convention the pathname of the system library file is /mrt/syslib.

user pathname
The file pathname is the process file of a process to be started up by the kernel initialization process at boot time. Only one pathname can be specified with each user keyword. The user keyword may be repeated nine times.

The following key words are associated with construction of low core and memory management tables. These specifications are ignored if the "-u" option is specified.

memory start size [start size [start size]]
Physical memory is broken up into one, two, or three partitions. Start is the beginning (64 byte) block address of the partition,

size is the number of (64 byte) blocks in the partition. At system initialization time the actual size of memory is determined and the size of the last partition is adjusted to reflect the top of memory.

messages n
N * 16 word message buffers are allocated. The default is 32 and the maximum allowed is 160.

nrsde n
N resident segment descriptor entries (RSDE) will be allocated in the kernel private data segment (not low core). The number of RSDEs determines the maximum number of segments that can exist in the system at any time. One should allow about 3.7 segments per process.

processes n
N process table entries (DCT) are allocated. The default is 50 and the maximum is 127.

stack size
The size of the system stack is size bytes. The system stack resides at the high address end of the lowcore segment.

v addr csr
the v (for vector) keyword is included for defining non-standard devices or standard devices which use interrupt vectors and/or control and status registers which do not conform with DEC conventions. Addr is the interrupt vector address used by the device and csr is the address of the device control and status register (the register containing the interrupt enable bit).

The following keywords are provided to handle standard DEC devices and are simply "built in" v keywords (e.g. the program knows the vector and csr addresses).

console	pcl1	kwllp	parity	plot
ad01	afcl1	aalld	aalll	lpl1
lsl1	rfl1	rcl1	tcl1	tml1
tul1	htl1	rkl1	cdl1	cml1
crl1	udcl1	rpl1	rp03	hpl1
rjp04	tfl1	tal1	dcl1	kl11
dl11a	dl11b	dl11c	dl11d	dl11e
dm11a	dm11b	dn11	dpl1	dr11a
dr11c	dtl1	djl1	dhl1	dq11
dul1				

FILES

/bin/ld
/bin/as
lcor0.s generated by sgen and used to form lowcore image
lcor1.s contains constants which must be in I = D = phyical memory

ALSO SEE
ldp(e), ll(e)

NAME

tdmp - dump system state into core file

SYNOPSIS

tdmp [korefile]

DESCRIPTION

Tdmp reads the state of the system as it was saved on DECtape (drive 1) when the system "panic'ed" producing a dump of all of memory starting at block 0 on the DECtape. If no korefile is specified, the dump is put in the file "kore" in the current directory.

The "kore" file is suitable for debugging with kdb.

SEE ALSO

kdb(e), kdmp(e).

BUGS

INTRODUCTION TO MERT UNIX SYSTEM CALLS

Section F of this manual lists all of the additional entries into the UNIX system over and above those described in section II. These entries are only supported by the MERT system. These additional entries into the UNIX system can be categorized as follows:

(1) support of multi-environment features
(2) physical and asynchronous I/O into user area
(3) use of system message facilities
(4) sharing of segments between processes

In most cases two calling sequences are specified, one of which is usable from assembly language, and the other from C. Most of these calls have an error return. From assembly language an erroneous call is always indicated by turning on the c-bit of the condition codes. The presence of an error is most easily tested by the instructions bes and bec (``branch on error set (or clear)''). These are synonyms for the bcs and bcc instructions.

From C, an error condition is indicated by an otherwise impossible returned value. Almost always this is -1; the individual sections specify the details.

In both cases an error number is also available. In assembly language, this number is returned in r0 on erroneous calls. From C, the external variable errno is set to the error number. Errno is not cleared on succesful calls, so it should be tested only after an error has occurred. There is a table of messages associated with each error, and a routine for printing the message. See perror (III).

The possible error numbers are not recited with each writeup in section F, since many errors are possible for most of the calls. See the introduction to section II for a list of all the possible error numbers.

NAME

call - combination fork, exec, wait

SYNOPSIS

```
(call = 70.)
sys call; &status; name; args
...
status: .=.+2
...
name: <...\0>
...
args: arg1; arg2; ...; 0
arg1: <...\0>
arg2: <...\0>
   ...

lcall(&status, name, arg1, arg2, ..., argn, 0)
int status;
char *name, *arg1, *arg2, ..., *argn;

vcall(&status, name, argv)
int status;
char *name;
char *argv[ ];
```

DESCRIPTION

Call does the equivalent of a fork and a wait for the parent process and the equivalent of a exec for the child process which is spawned. Call spawns a child process without making a copy of the user segments; the child process sets up its user address space with the named file and then transfers to the beginning of the core image of the file.

Files remain open across call calls. Ignored signals remain ignored across call, but signals that are caught are reset to their default values.

Each user has a real user ID and group ID and an effective user ID and group ID (The real ID identifies the person using the system; the effective ID determines his access privileges.) Call changes the effective user and group ID to the owner of the executed file if the file has the ``set-user-ID'' or ``set-group-ID'' modes. The real user ID is not affected.

The form of this call differs somewhat depending on whether it is called from assembly language or C; see below for the C version.

The first argument to call is the address of the status word in which the result of the call is returned. The second argument to call is a pointer to the name of the file to be executed. The third is the address of a null-terminated list of pointers to arguments to be passed to the file. Conventionally, the second argument is the name of the file.

Each pointer addresses a string terminated by a null byte.

Cnce the called file starts execution, the arguments are available as follows. The stack pointer points to a word containing the number of arguments. Just above this number is a list of pointers to the argument strings. The arguments are placed as high as possible in core.

```
 sp-> nargs
      argl
      ...
      argn

argl: <argl\0>
      ...
argn: <argn\0>
```

From C, two intefaces are available. lcall is useful when a known file with known arguments is being called; the arguments to lcall are the character strings constituting the file and the arguments; as in the basic call, the first argument is conventionally the same as the file name (or its last component). A 0 argument must end the argument list.

The vcall version is useful when the number of arguments is unknown in advance; the arguments to vcall are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

When a C program is executed, it is called as follows:

```
main(argc, argv)
int argc;
char *argv[];
```

where argc is the argument count and argv is an array of character pointers to the arguments themselves. As indicated, argc is conventionally at least one and the first member of the array points to a string containing the name of the file.

Argv is not directly usable in another vcall, since argv[argc] is -1 and not 0.

SEE ALSO

fork(II), exec(II), wait(II)

DIAGNOSTICS

If the file cannot be found, if it is not executable, if it does not have a valid header (407, 410 or 411 octal as first word), if maximum memory is exceeded, if the arguments require more than 512 bytes, or if a new process cannot be created, the error bit (c-bit) is set. From C the returned value is -1.

BUGS

Only 512 characters of arguments are allowed.

NAME

falloc - allocate space for contiguous file

SYNOPSIS

```
(falloc = 77.)
sys falloc; name; mode; size[0]; size[1]
(file descriptor in r0)

falloc(name, mode, size[0], size[1])
char *name;
```

DESCRIPTION

Falloc creates a new file called name, given as the address of a null-terminated string. The file must not already exist. It is given mode mode and the ICONT bit is also set in the inode (see chmod(c)). Contiguous file system space is allocated to the file corresponding to the number of bytes specified by size[0] (high order size word) and size[1] (low order size word). 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 the actual size of the file.

The file is also opened for writing, and its file descriptor is returned in r0.

This system call is typically used to insure the allocation of contiguous file system space for a large file which will be described by one extent. Large physical or asynchronous I/O transfers may be issued to these files with the assurance of good real-time response.

SEE ALSO

creat(II), creat(c), chmod(c).

DIAGNOSTICS

The error bit (c-bit) is set if a needed directory is not searchable, the file already exists, there is insufficient file system space or too many files are already open. From C, a -1 value is returned on an error.

NAME

fmove - move file into contiguous area

SYNOPSIS

(fmove = 78.)
sys fmove; dev; inode; flag

fmove(dev, inode, flag)

DESCRIPTION

Fmove moves the file specified by dev which is the device number of the file system on which it resides and by inode which is the file number on that file system into a contiguous area of the file system starting at a lower block address if possible. The arguments for fmove are typically returned by fstat or stat (see fstat(II) and stat(II)). The file is locked and may not be accessed by other users while the file is being moved. If the value of flag is zero and the file cannot be moved into a contiguous area lower than which it currently resides in, it is not moved. If flag is non-zero, the file will always be moved if enough contiguous space is available.

This system call may be used to move a file into a contiguous area thereby guaranteeing minimum latency in read/write operations on the file and ensuring fast response to large physical I/O transfers to/from the file.

SEE ALSO

fmove(c), fmove(d).

DIAGNOSTICS

The error bit (c-bit) is set if the user is not super-user, the file is currrently active or it cannot be moved because of insufficient space. From C, a -1 value is returned on an error.

NAME

getseg - get user segment

SYNOPSIS

```
(getseg = 71.; not in assembler)
(function code in r0)
sys getseg; segstruct
(code in r0)

segd = getseg(segstrp)
code = rmovseg(segstrp)
segd = connseg(segstrp)
code = discseg(segstrp)
segd = getsnam(segstrp)
struct segstruct *segstrp;
```

DESCRIPTION

The five routines manipulate user segments in the user's address space (either I or D). The user may have up to six (6) segments in his address space in addition to the code, data and stack segments at any one time. The user can manipulate more than six segments at a time but only six of these may be in his active address space simultaneously.

From assembly language, the function code in register r0 specifies the request type.

0 The user may put a segment known by name in his address space or a new segment may be created in his address space.
1 A segment may be completely removed from the user's address space and the segment will no longer be one of his available segments.
2 An already existing segment (previously put in his address space by a call to getseg) may be put in the user's active address space.
3 A segment may be disconnected from the user's address space. This segment is still available to be put in his active address space at a later time.
4 The name of a user segment may be determined. The segment may be a part of the user's address space as the result of the inclusion of a public library.

All uses of this system call require the reference to a five-word (5) structure as follows:

```
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 code, 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 segd equal to 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.

Getseg allocates a segment in the user's address space. If the name, segname is zero, the segment is assumed not to exist and a new one is created. However, if the name is non-zero, it is assumed to already exist. The base register to be used to point to the user segment may be specified by breg. A value of breg from 0 to 7 will put the segment in the I-mode address space of the user; a value from 8 to 15 will put the segment in the D-mode address space of the user. If the user wishes the system to determine the next available base register for the segment, a value of -1 must be given for breg. The address of the logical segment will start on a segment boundary, i.e. a multiple of 8192 bytes and will be segsize bytes in length. The user must specify the length of the segment to be created; however, if the segment already exists and the user has been passed the name of the segment in a message, a length of zero bytes should be specified to ensure that a brand new segment is not created. This segment may be shared between the parent and any number of its children. The share permissions of the segment perm are specified for the child in bits <5-3> and for the parent in bits <2-0> as follows:

<5-3> - 0 - not shared
2 - read only
6 - read/write
7 - shareable by process created by "pcreat"
<2-0> - 2 - read only
6 - read/write.

A user segment number (descriptor) is returned in segd. The name of the segment is returned in segname and may be used for future reference to the segment. The size of the segment in bytes is returned in segsize in case the size was not known. In all cases the starting address of the segment is returned in segaddr. This segment is made one of the user's active segments. The shared segment is maintained across all system fork's and exec's. Synchronization may be achieved by means of the sendev and waitev primitives.

Rmovseg is used to remove a segment completely from the user's address space. The slot in the user's process PCB is also freed up for later use. All that the user need specify in the structure is the segment name. The name may be either a 32-bit name or a number less than 256, in which case it is treated as a segd. The segment can no longer be put back in the user's address space, unless another process still has it as one of its segments and the user invokes getseg again with the name of the segment. A positive re-

turn code indicates success, i.e. the segment did exist and was in fact removed.

Connseg connects a segment in the process's PCB table into the user's address space at the specified address. If the value of breg is -1, the next available user base register will be used, other wise the one specified by breg will be used. The name of the segment must be passed in segname. If the name is less than 256, it is taken to be the segment descriptor, segd. The segment descriptor, segd is returned in r0. The starting address of the user segment is returned in segaddr.

Discseg disconnects a user segment from his active address space. The name specified has the same convention as specified by connseg above. The segment may be connected as an active segment at a later time. A positive return code indicates a successful function call.

Getsnam returns the name of the segment specified by segd in the segment name field, segname. The 32-bit name of the segment is returned in segname. The segment descriptor is returned in r0. The segment is not put in the user's active address space.

SEE ALSO

sendev(f), waitev(f).

DIAGNOSTICS

The error bit (c-bit) is set if a new segment cannot be created because of insufficient swap space. From C, a -1 value is returned on an error. Other error conditions possible are the non-existence of the named segment or insufficient address space in the user's address space for the segment.

NAME

lock - lock a flag
unlock - unlock a flag
tlock - test and lock a flag

SYNOPSIS

```
(lock = 80.)
sys lock; func; flag
(current value of flag in r0)
          func = 0 lock
                 1 unlock
                 2 tlock

lock(flag)
unlock(flag)
tlock(flag)
```

DESCRIPTION

Lock suspends execution of the calling process while the semaphore is locked by another process. Otherwise the semaphore is locked by setting the flag word to the current process ID. If the current value of flag is non-zero, the process sleeps on the address of the flag. Unlock sets the value of flag to 0 and wakes up all processes waiting on this flag. Tlock sets the value of the flag to the current process ID only if the current value of the flag is zero. The calling process is not suspended.

SEE ALSO

DIAGNOSTICS

The error bit (c-bit) is set if an illegal function is requested. From C, a -1 value is returned on an error.

NAME

msgport - send message to a process connected to a port

SYNOPSIS

```
(msgport = 69.)
(message size in r0)
sys msgport; msgbuf

msgport(msgbuf, msgsize)
```

DESCRIPTION

Msgport sends the message in the message buffer msgbuf to the process connected to the port specified in the msto word of the message header (see sendmsg(c) for message header format). The size of the message to be sent (in words) msgsize includes the six word message header. The sender must fill in the msto word of the message header. The system fills in the mssize byte and the mstype byte. The user is only allowed to send type -3 messages. The acknowledgements to the sender's messages may be identified by filling in the msident word in the message header.

SEE ALSO

msgsend(f), sendmsg(c), getmsg(c), msgrecv(f).

DIAGNOSTICS

The error bit (c-bit) is set if the intended receiver process is non-existent, the port number is invalid, no process is connected to the port, or if the message buffer is not in the user's address space. From C, a -1 value is returned on an error.

NAME

msgrecv - receive message

SYNOPSIS

```
(msgrecv = 66.)
(message size in rØ)
sys msgrecv; msgbuf

msgrecv(msgbuf, msgsize)
```

DESCRIPTION

Msgrecv receives a message into the message buffer msgbuf (see sendmsg(c) for message header format). The maximum size of the message that will be received (in words) is msgsize including the six word message header. The system fills in the mssize byte and the mstype byte. The mssize byte will contain the actual number of words received. The user will only receive type -3 messages. Acknowledgements to particular messages may be identified by the msident word in the message header.

The status byte in the message header may be used by the sending process to indicate the status of the particular message request.

SEE ALSO

sendmsg(c), getmsg(c), msgsend(f).

DIAGNOSTICS

The error bit (c-bit) is set if the message buffer address is not in user's address space. From C, a -1 value is returned on an error.

NAME

msgsend - send message to a process

SYNOPSIS

```
(msgsend = 67.)
(message size in rØ)
sys msgsend; msgbuf

msgsend(msgbuf, msgsize)
```

DESCRIPTION

Msgsend sends the message in the message buffer msgbuf to the process specified in the msto word of the message header (see sendmsg(c) for message header format). The size of the message to be sent (in words) msgsize includes the six word message header. The sender must fill in the msto word of the message header. The system fills in the mssize byte and the mstype byte. The user is only allowed to send type -3 messages. The acknowledgements to the sender's messages may be identified by filling in the msident word in the message header.

SEE ALSO

sendmsg(c), getmsg(c), msgrecv(f).

DIAGNOSTICS

The error bit (c-bit) is set if the intended receiver process is non-existent or if the message buffer is not in the user's address space. From C, a -1 value is returned on an error.

NAME

pcreat - creat new process

SYNOPSIS

```
(pcreat = 65.)
sys pcreat; name

pcreat(name)
char *name;
```

DESCRIPTION

Pcreat sends off a message to the process manager to create a new process from the description given in the file name. Name is the address of a string of ASCII characters representing a path name, terminated by a null character. A user segment which has been created by using connseg (see connseg(f)) with child permissions equal to 07 may be shared with the newly created process.

The process number of the new process is returned in r0. The process may be "waited" for using the standard "sys wait" system entry. This system entry is used to set up a new environment other than the UNIX time-sharing environment. The file containing the initial process image is created using ldp (see section E).

SEE ALSO

fork(II), wait(II), ldp(e).

DIAGNOSTICS

The error bit (c-bit) is set if the file does not exist, the file is not in the correct format or a new process could not be created because of lack of process space. From C, a -1 value is returned on an error.

NAME

plock - lock process in memory

SYNOPSIS

```
(plock = 79.)
(lock flag in r0)
sys plock

plock(flag)
```

DESCRIPTION

Plock locks the current process in memory if the value of flag is non-zero and unlocks the current process if flag is zero. When a process is locked in memory, all segments belonging to the process are marked as non-swappable. These segments may be shifted in memory but may not be written back to secondary.

SEE ALSO

pswap(a), punswap(a).

DIAGNOSTICS

The error bit (c-bit) is set if less than 8K words of memory are available for swapping after locking all of a process's segments in memory. From C, a -1 value is returned on an error.

NAME

sendev - send event(s)

SYNOPSIS

```
(sendev = 73.)
(process number in r0)
sys sendev; event

sendev(proc, event)
```

DESCRIPTION

Sendev sends any number of events (up to 8 maximum) to the process proc. The eight possible event flags are the low eight bits of the word event.

This primitive may be used to synchronize the execution of co-operating processes sharing a common data segment.

SEE ALSO

connseg(f), waitev(f).

DIAGNOSTICS

The error bit (c-bit) is set if the process proc does not exist. From C, a -1 value is returned on an error.

NAME

setio - set I/O mode of file

SYNOPSIS

```
(setio = 75.)
(file descriptor in r0)
sys setio; mode

setio(fildes, mode)
```

DESCRIPTION

Setio sets the I/O mode of subsequent reads or writes to the file specified by the file descriptor fildes which is a word returned from a successful open on a file. The possible modes which may be set for file I/O are:

- 0 - normal buffered I/O
- 01 - physical I/O directly to or from the user's address space
- 02 - asynchronous I/O directly to or from the user's address space

If asynchronous I/O to the user's address space is initiated, the I/O must eventually be waited for by a call to statio. To obtain physical I/O without system side-buffering, a file descriptor of -1 may be specified with a mode of 1. A setio(-1,0) will turn off the general physical I/O mode and revert back to normal system side-buffering.

In the case of physical I/O, if the I/O does not start on a device block boundary, i.e. a multiple of 256 words, normal system side-buffering is used. An I/O transfer may be broken up into a combination of physical I/O and buffered I/O by the system if this is possible. In the case of asynchronous I/O to or from the user's address space, the transfer must always start on a device block boundary but need not necessarily end on a block boundary.

SEE ALSO

open(II), read(II), write(II), read(c), write(c), statio(f).

DIAGNOSTICS

The error bit (c-bit) is set if the file descriptor is not that of an open file. From C, a -1 value is returned on an error.

NAME

statio - get status of asynchronous I/O

SYNOPSIS

```
(statio = 76.)
(buffer descriptor in r0)
sys statio; statbuf; wflag

statio(bufdes, statbuf, wflag)
int statbuf[3]; /* buffer descriptor */
                /* flag word indicating I/O status */
                /* I/O byte count */
```

DESCRIPTION

Statio returns the status of the asynchronous I/O transfer initiated by a read or write from or to a file for which the I/O mode had previously been set to asynchronous I/O (see setio(f)). The read or write returns immediately with a system I/O buffer descriptor bufdes which must be saved for later reference to the status of this I/O transfer.

To check the status of the I/O transfer a call to statio with a zero value of wflag will return immediately with the status of the I/O in the second word of the status buffer and the I/O byte count in the third word of the status buffer. The flags in the flag word are as follows:

02 - I/O complete
04 - I/O error

If the I/O transfer is not complete, computation may be resumed and I/O completion checked for at a later time. A statio call with a non-zero value of wflag will not return control to the user until this particular I/O is complete. A statio call with bufdes equal to zero will return the I/O status of the first outstanding asynchronous I/O completed. Currently a total of four asynchronous I/O transfers may be initiated at any one time on up to four different files.

SEE ALSO

open(II), read(II), write(II), read(c), write(c), setio(f).

DIAGNOSTICS

The error bit (c-bit) is set if the buffer descriptor is not legal or if there is no outstanding I/O waiting to be completed. From C, a -1 value is returned on an error.

NAME

sysproc - system process

SYNOPSIS

```
(sysproc = 68.)
(process port number in r0)
sys sysproc; flag

process = sysproc(syspnum, flag)
```

DESCRIPTION

Sysproc performs some operation on one of the system process ports specified by syspnum according to the value of the flag flag. Syspnum is a value from 0 to the maximum process port number (currently 4). If flag is 0 the current process is connected to the process port syspnum. If flag is 1 the current process is disconnected from the process port syspnum. If flag is 2 the value of the process process connected to the process port syspnum is returned.

SEE ALSO

msgport(f).

DIAGNOSTICS

The error bit (c-bit) is set if the port number or flag value is invalid. From C, a -1 value is returned on an error.

NAME

waitev - wait for an event

SYNOPSIS

```
(waitev = 74.)
sys waitev
(event word in r0)

event = waitev()
```

DESCRIPTION

Waitev waits for the receipt of an event from a process. The eight possible event flags are the low eight bits of the word event.

This primitive may be used to synchronize the execution of co-operating processes sharing a common data segment.

SEE ALSO

connseg(f), sendev(f).

DIAGNOSTICS