NAME

mpx, join, chan, extract, attach, detach, connect, npgrp, ckill, mpxcall − create and manipulate multiplexed files

SYNOPSIS

mpx (name, access)
char *name;

join (fd, xd)

chan (xd)

extract (i, xd)

attach (i, xd)

detach (i, xd)

connect (fd, cd, end)

npgrp (i, xd, pgrp)

ckill (i, xd, signal)

#include <sys/mx.h>
mpxcall (cmd, vec)
int *vec;

DESCRIPTION

Mpxcall (cmd, vec) is the system call shared by the library routines described below. *Cmd* selects a command using values defined in **<sys/mx.h>**. *Vec* is the address of a structure containing the arguments for the command:

mpx (name, access)

Mpx creates and opens the file *name* with access permission *access* (see *creat* (2)) and returns a file descriptor available for reading and writing. A −1 is returned if the file cannot be created, if *name* already exists, or if the file table or other operating system data structures are full. The file descriptor is required for use with other routines.

If *name* designates a null string, a file descriptor is returned as described, but no entry is created in the file system.

Once created, an mpx file may be opened (see *open* (2)) by any process. This provides a form of interprocess communication whereby a process B can "call" process A by opening an mpx file created by A. To B, the file is ordinary with one exception: the *connect* primitive could be applied to it. Otherwise, the functions described below are used only by process A and descendants that inherit the open mpx file.

When a process opens an mpx file, the owner of the file receives a control message when the file is next read. The method for "answering" this kind of call involves using *attach* and *detach* as described in more detail below.

Once B has opened A's mpx file, it is said to have a *channel* to A. A channel is a pair of data streams: in this case, one from B to A and the other from A to B. Several processes may open the same mpx file, yielding multiple channels within the one mpx file. By accessing the appropriate channel, A can communicate with B and any others. When A reads (see *read* (2)) from the mpx file, data written to A by the other processes appears in A's buffer using a record format described in *mpxio* (5). When A writes (see *write* (2)) on its mpx file, the data must be formatted in a similar way.

The following commands are used to manipulate mpx files and channels.

join − adds a new channel on an mpx file to an open file F. I/O on the new channel is

I/O on F.
chan − creates a new channel.
extract − file descriptor maintenance.
connect − similar to join except that the open file F is connected to an existing channel.
attach and *detach* − used with call protocol.
npgrp − manipulates process group numbers so that a channel can act as a control terminal (see *tty*(4)).
ckill − send signal (see *signal*(2)) to process group through channel.

A maximum of 15 channels may be connected to an mpx file. They are numbered 0 through 14. *Join* may be used to make one mpx file appear as a channel on another mpx file. A hierarchy or tree of mpx files may be set up in this way. In this case, one of the mpx files must be the root of a tree where the other mpx files are interior nodes. The maximum depth of such a tree is 4.

An *index* is a 16-bit value that denotes a location in an mpx tree other than the root: the path through mpx "nodes" from the root to the location is expressed as a sequence of 4-bit nibbles. The branch taken at the root is represented by the low-order 4-bits of an index. Each succeeding branch is specified by the next higher-order nibble. If the length of a path to be expressed is less than 4, then the illegal channel number, 15, must be used to terminate the sequence. This is not strictly necessary for the simple case of a tree consisting of only a root node; its channels can be expressed by the numbers 0 through 14. An index *i* and file descriptor *xd* for the root of an mpx tree are required as arguments to most of the commands described below. Indices also serve as channel identifiers in the record formats given in *mpxio*(5). Since −1 is not a valid index, it can be returned as a error indication by subroutines that normally return indices.

The operating system informs the process managing an mpx file of changes in the status of channels attached to the file by generating messages that are read along with data from the channels. The form and content of these messages is described in *mpxio*(5).

join (fd, xd) establishes a connection (channel) between an mpx file and another object. *Fd* is an open file descriptor for a character device or an mpx file and *xd* is the file descriptor of an mpx file. *Join* returns the index for the new channel if the operation succeeds and −1 if it does not.

Following join, *fd* may still be used in any system call that would have been meaningful before the join operation. Thus, a process can read and write directly to *fd* as well as access it via *xd*. If the number of channels required for a tree of mpx files exceeds the number of open files permitted a process by the operating system, some of the file descriptors can be released using the standard *close*(2) call. Following a close on an active file descriptor for a channel or internal mpx node, that object may still be accessed through the root of the tree.

chan (xd) allocates a channel and connects one end of it to the mpx file represented by file descriptor *xd*. *Chan* returns the index of the new channel or a −1 indicating failure. The *extract* primitive can be used to get a non-multiplexed file descriptor for the free end of a channel created by *chan*.

Both *chan* and *join* operate on the mpx file specified by *xd*. File descriptors for interior nodes of an mpx tree must be preserved or reconstructed with *extract* for use with *join* or *chan*. For the remaining commands described here, *xd* denotes the file descriptor for the root of an mpx tree.

Extract (i, xd) returns a file descriptor for the object with index *i* on the mpx tree with root file descriptor *xd*. A −1 is returned by extract if a file descriptor is not available or if the arguments do not refer to an existing channel and mpx file.

attach (i, xd)
detach (i, xd). If a process A has created an mpx file represented by file descriptor *xd*, then a

process B can open (see *open*(2)) the mpx file. The purpose is to establish a channel between A and B through the mpx file. *Attach* and *Detach* are used by A to respond to such opens.

An open request by B fails immediately if a new channel cannot be allocated on the mpx file, if the mpx file does not exist, or if it does exist but there is no process (A) with a multiplexed file descriptor for the mpx file (i.e. *xd* as returned by *mpx*(2)). Otherwise, a channel with index number *i* is allocated. The next time A reads on file descriptor *xd*, the WATCH control message (see *mpxio*(5)) will be delivered on channel *i*. A responds to this message with *attach* or *detach*. The former causes the open to complete and return a file descriptor to B. The latter deallocates channel *i* and causes the open to fail.

One mpx file may be placed in 'listener' mode. This is done by writing *ioctl(xd, MXLSTN, 0)* where *xd* is an mpx file descriptor and MXLSTN is defined in **/usr/include/mx.h**. The semantics of listener mode are that all file names discovered by *open*(2) to have the syntax **system!pathname** (see *uucp*(1C)) are treated as opens on the mpx file. The operating system sends the listener process an OPEN message (see *mpxio*(5)) which includes the file name being opened. *Attach* and *detach* then apply as described above.

Detach has two other uses: it closes and releases the resources of any active channel it is applied to, and should be used to respond to a CLOSE message (see *mpxio*(5)) on a channel so the channel may be reused.

connect(fd, cd, end). *Fd* is a character file descriptor and *cd* is a file descriptor for a channel, such as might be obtained via *extract(chan(xd), xd)* or by *open*(2) followed by *attach*. *Connect* splices the two streams together. If *end* is negative, only the output of *fd* is spliced to the input of *cd*. If *end* is positive, the output of *cd* is spliced to the input of *fd*. If *end* is zero, then both splices are made.

npgrp(i, xd, pgrp). If *xd* is negative, *npgrp* applies to the process executing it, otherwise *i* and *xd* are interpreted as a channel index and mpx file descriptor, and *npgrp* is applied to the process on the non-multiplexed end of the channel. If *pgrp* is zero, the process group number of the indicated process is set to the process number of that process, otherwise the value of *pgrp* is used as the process group number.

Npgrp normally returns the new process group number. If *i* and *xd* specify a nonexistant channel, *npgrp* returns −1.

ckill(i, xd, signal) sends the specified signal (see *signal*(2)) through the channel specified by *i* and *xd*. If the channel is connected to anything other than a process, *ckill* is a null operation. If there is a process at the other end of the channel, the process group will be interrupted (see *signal*(2), *kill*(2)). *Ckill* normally returns *signal*. If *ch* and *xd* specify a nonexistent channel, *ckill* returns −1.

FILES

/usr/include/sys/mx.h
/usr/include/sys/ioctl.h

SEE ALSO

ioctl(2), mpxio(5)

BUGS

Mpx files are an experimental part of the operating system more subject to change and prone to bugs than other parts. Maintenance programs, e.g. *icheck*(1M), diagnose mpx files as an illegal mode. Channels may only be connected to objects in the operating system that are accessible through the line discipline mechanism. Higher performace line disciplines are needed. A non-destructive *disconnect* primitive (inverse of *connect*) is not provided. A non-blocking flow control strategy based on messages defined in *mpxio*(5) should not be attempted by novices; the enabling *ioctl* command should be protected. The *join* operation could be subsumed by *connect*. A mechanism is needed for moving a channel from one location in an mpx tree to another.

Attempts to read when there are less than 14 bytes left in the channel buffer may cause an incorrect length to be returned on the read. There are problems with splicing channels and redirecting them.

NAME

msg, msgenab, msgdisab, send, sendw, recv, recvw, msgstat, msgctl − send and receive messages

SYNOPSIS

```
# include <sys/ipcomm.h>
msgenab ( )
msgdisab ( )

send (buf, size, topid, type)
sendw (buf, size, topid, type)
char *buf;

recv (buf, size, &mstructp, type)
recvw (buf, size, &mstructp, type)
char *buf;
struct mstruct mstructp;

msgstat (&mstat, sizeof (mstat), pid)
struct mstat mstat;

msgctl (pid, command, arg)
```

DESCRIPTION

A process that has enabled message reception has a message queue on which are placed, in order of arrival, messages sent to it by other processes. The process actually receives a message's contents by requesting a message from the queue. A process may send a message to any other process that has enabled message reception, as long as the receiver does not have an excessive number of messages pending on its queue.

From assembly language, the *function* argument specifies the request type.

0 Message reception is disabled; messages may no longer be sent to the process. Depending on the *type*, any message(s) still on the queue are either discarded or returned to the sender. No other arguments are used for this kind of request.

1 Enable message reception. No messages may be sent to the process until this is done. No other arguments are used in this kind of request. Message reception remains enabled across *exec*, but not across *fork*.

2 Send a message to another process. If the system's message buffers are temporarily full, return is immediate. (Conditional Send)

3 Send a message to another process. This is as above, except that execution may be suspended until there is sufficient buffer space to send the message. (Unconditional Send)

4 Receive the first message on the queue of the requested *type*. Return immediately if no such message exists. (Conditional Receive)

5 Receive a message as above, except that execution may be suspended until a suitable message is placed on the queue, if one is not already available. (Unconditional Receive)

6 Request a count of the number of messages allowed and actual number of messages queued for the process numbered *pid*.

7 Set control variables in message queue header as defined by *command*. At present, only available command is *setmqlen* which sets maximum number of messages allowed by process numbered *pid*.

The *buf* argument is the address of the buffer that, when sending, contains the message to be sent, or, when receiving, is where the message is to be placed. The number of bytes to be sent or received should be in r0. Currently, messages may be from 0 to 212 bytes in length. If, when receiving, the length of the message exceeds the requested number of bytes, the message is truncated. In any event, the number of bytes actually sent or received is returned in r0.

When a message is being sent, *arg3* should contain the processid of the receiving process. When receiving a message, *arg3* should be the address of a structure of type mstruct.

The *type* argument is used by a sender to assign a type number (1 to 128) to a message. By convention, types 1 to 63 imply that an acknowledgement message is desired; types 64 to 128 imply no acknowledgement is necessary; type 128 is an acknowledgement message. If a process disables messages (or exits) with any messages still on its queue, those of type 1 to 63 are changed to type 128 and, if possible, returned to the sender; those of type 64 to 128 are discarded.

When receiving messages, a process may request *type* 0, indicating that the first message on the queue is to be retrieved, or a *type* from 1 to 128, indicating that the first message on the queue of the requested *type* is to be received. In either case, the message's actual type is returned in the second word of the structure provided by the user *arg3*.

From C, *msgenab* and *msgdisab* enable and disable message reception, respectively. *Msgstat* returns message status in terms of actual and maximum allowed message queue lengths. *Msgctl* allows modification of the maximum number of messages parameter. All return zero when successful.

The *send*, *sendw*, *recv*, and *recvw* functions perform conditional send, unconditional send, conditional receive, and unconditional receive operations, respectively. All return the number of bytes actually sent or received, as appropriate. The format of *ipcomm.h* is as follows:

```
/*              %W%             */

/*
 * Interprocess Communication Control Structures
 */

#ifdef KERNEL
/*
 * common flags
 */

#define IP_PERM         03                      /* scope permission mask */
#define IP_ANY 0                        /* system scope */
#define IP_UID  01                      /* userid scope */
#define IP_GID  02                      /* groupid scope */
#define IP_QWANT        0100            /* entry in msg queue wanted */
#define IP_WANTED       0200            /* resource is desired */

struct ipaword
{               char            ip_flag;
                char            ip_id; };

/*
 * message control
 */

#define PMSG    5                       /* message sleep priority */
#define MAXMLEN         212                     /* max message length in bytes */
#define MAXMSGDEF       10                      /* default max number unreceived msgs per p
#define MAXMSGL         20                      /*max limit to be set by msgctl*/
```

```
#define MSGIO  02                                   /* tell iomove() this is msg */
#define MSGIN  0                                    /* same as B_WRITE */
#define MSGOUT                  01                          /* same as B_READ */

#define MDISAB                  0
#define MENAB 1
#define MSEND 2
#define MSENDW                  3
#define MRECV 4
#define MRECVW                  5
#define MSTAT 6
#define MSGCTL                  7

struct msghdr
{               struct msghdr   *mq_forw;
                int             mq_size;
                int             mq_sender;
                int             mq_type;
};
struct msgqhdr
{               struct msghdr   *mq_forw;       /* note same position as in msghdr */
                struct msghdr   *mq_last;
                int             *mq_procp;
                char            mq_flag;
                char            mq_cnt;
                int             mq_meslim;
};

#endif

/* commands for msgctl call here */
#define SETMQLEN 0                                  /*set mes q length command*/


struct mstat {
                unsigned        ms_cnt;
                unsigned        ms_maxm;
};

struct mstruct {
                int             ms_frompid;
                int             ms_type;
};
```

DIAGNOSTICS

The error bit (c-bit) is set for any one of a number of error conditions. An error occurs when enabling messages if no queue is available for use; it is also erroneous to attempt to disable message reception if it is not enabled. When trying to send messages, errors occur because the message is too long, the receiver has not enabled message reception, the type specified is not valid, the receiver has an excessive number of messages outstanding on its queue, or, for conditional sends, the system message buffers are temporarily full. When receiving messages, errors may occur because the process has not enabled message reception, the requested type or size are invalid, or, for conditional receives, a message of the requested type is not on the queue. It is also illegal to set the message limit (via *msgctl*) to a value larger than defined by MAXMXSGDEF in **ipcomm.h**. From C, a −1 return from any function indicates an error.

ASSEMBLER

(msg = 49.; not in assembler)
(size in r0)
sys msg; function; buf; arg3; type

FILES

/usr/include/sys/ipcomm.h

NAME

nice − set program priority

SYNOPSIS

nice (priority)

DESCRIPTION

The scheduling *priority* of the process is changed to the argument. Positive priorities get less service than normal; 0 is default. Only the super-user may specify a negative priority. The valid range of *priority* is **20** to **−128**. The value of 16 is recommended to users who wish to execute long-running programs without flak from the administration.

The effect of this call is passed to a child process by the *fork* system call. The effect can be cancelled by another call to *nice* with a *priority* of 0. From assembler, *nice* returns the old priority in r0. From C, *nice* returns a zero if no errors were encountered.

SEE ALSO

nice(1)

DIAGNOSTICS

The error bit (c-bit) is set if the user requests a *priority* outside the range of 0 to 20 and is not the super-user. In C, a −1 indicates an error.

ASSEMBLER

(nice = 34.)
(priority in r0)
sys nice
(old priority in r0)

NAME

open – open for reading or writing

SYNOPSIS

open (name, mode)
char *name;

DESCRIPTION

Open opens the file *name* for reading (if *mode* is *0*), writing (if *mode* is *1*) or for both reading and writing (if *mode* is *2*). *Name* is the address of a string of ASCII characters representing a path name, terminated by a null character.

The returned file descriptor should be saved for subsequent calls to *read*, *write*, *lseek*, *close*, etc.

Normally a single process may have as many as 20 files *open*ed simultaneously. The file descriptors returned will be in the range 0 to 19. To cause a file descriptor to be "auto-closed" on *exec*, use *ioctl*(2).

SEE ALSO

dup(2), creat(2), read(2), write(2), close(2), ioctl(2)

DIAGNOSTICS

The error bit (c-bit) is set if the file does not exist, if one of the necessary directories does not exist or is unreadable, if the file is not readable (resp. writable), or if 20 files are open. From C, a –1 value is returned on an error.

ASSEMBLER

(open = 5.)
sys open; name; mode

NAME

pause – stop until signal

SYNOPSIS

pause ( )

DESCRIPTION

Pause is used to give up control while waiting for a signal from *kill*(2) or *alarm*(2). It only returns control if a signal is raised and not ignored, and control returns from the signal action routine.

A pause without an alarm having previously been set will return immediately with a zero value.

SEE ALSO

kill(1), kill(2), alarm(2), signal(2), setjmp(3C)

ASSEMBLER

(pause = 29.)

sys pause

NAME

pipe − create a pipe

SYNOPSIS

pipe (fildes)
int fildes[2];

DESCRIPTION

The *pipe* system call creates an I/O mechanism called a pipe. The file descriptors returned can be used in read and write operations. When the pipe is written using the descriptor returned in r1 (resp. *fildes*[1]), up to 4096 bytes of data are buffered before the writing process is suspended. A read using the descriptor returned in r0 (resp. *fildes*[0]) will pick up the data.

It is assumed that after the pipe has been set up, two (or more) cooperating processes (created by subsequent *fork* calls) will pass data through the pipe with *read* and *write* calls.

The shell has a syntax to set up a linear array of processes connected by pipes.

Read calls on an empty pipe (no buffered data) with only one end (all write file descriptors closed) return an end-of-file. *Write* calls under similar conditions are ignored.

SEE ALSO

sh(1), read(2), write(2), fork(2)

DIAGNOSTICS

The error bit (c-bit) is set if there are not 2 free file descriptors when the *pipe* call is made. From C, a −1 returned value indicates an error.

ASSEMBLER

(pipe = 42.)
sys pipe
(read file descriptor in r0)
(write file descriptor in r1)

NAME

plock – lock process or text in memory

SYNOPSIS

plock (operation)

DESCRIPTION

A process can lock in memory its complete process image or just its text image, and thereby, have it immune to all routine swapping. The caller must be super-user. The argument *operation* specifies:

0 – PUNLOCK – remove locked status on process

1 - PROCLOCK – lock process text & data in memory

2 - TXTLOCK – lock only the text in memory

3 - TUNLOCK – remove locked status on text

Locked processes and texts are shuffled down to the lowest possible address in user swappable memory. *Locks* are not inherited by children across *forks* and *execs* by locked processes are illegal. *Locked* processes can still be swapped under certain circumstances such as those in the following warning.

WARNING:

A great deal of swapping, including the swapping of other locked processes, occurs whenever a process locks its text and/or data; or a locked process grows its data or stack, exits, or is the last locking process to free a locked text. Consequently, processes performing locking should possess long term stability. If the application of locking is to improve real time response, then the careless use of it will do more harm than good.

FILES

/usr/include/sys/lock.h

DIAGNOSTICS

The error bit (c-bit) is set if the proper lock-unlock sequence is not performed in order. i.e. locking the text then locking the process before unlocking the text is illegal.

ASSEMBLER

(syscb = 45.; lock = 3.)
(lock in r1)
sys syscb; operation

NAME

profil − execution time user profile

SYNOPSIS

profil (buff, bufsiz, offset, scale)
char *buff;
int bufsiz, offset, scale;

DESCRIPTION

Buff points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter (pc) is examined each clock tick (60th second); *offset* is subtracted from it, and the result multiplied by *scale*. If the resulting number corresponds to a word inside *buff*, that word is incremented.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0177777(8) gives a 1-1 mapping of pc's to words in *buff*; 077777(8) maps each pair of instruction words together. 02(8) maps all instructions onto the beginning of *buff* (producing a non-interrupting core clock).

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an *exec* is executed, but remains on in child and parent both after a *fork*. Profiling may be turned off if an update in *buff* would cause a memory fault.

SEE ALSO

prof(1), monitor(3C)

ASSEMBLER

(profil = 44.)
sys profil; buff; bufsiz; offset; scale

NAME

ptrace – process trace

SYNOPSIS

ptrace (request, pid, addr, data)
int request, pid, addr, data;

DESCRIPTION

Ptrace provides a means by which a parent process may control the execution of a child process, and examine and change its core image. Its primary use is for the implementation of breakpoint debugging, but it should be adaptable for simulation of non-UNIX environments. There are four arguments whose interpretation depends on a *request* argument. Generally, *pid* is the process ID of the traced process, which must be a child (no more distant descendant) of the tracing process. A process being traced behaves normally until it encounters some signal whether internally generated like "illegal instruction" or externally generated like "interrupt." See *signal*(2) for the list. Then the traced process enters a stopped state and its parent is notified via *wait*(2). When the child is in the stopped state, its core image can be examined and modified using *ptrace*. If desired, another *ptrace* request can then cause the child either to terminate or to continue, possibly ignoring the signal.

The value of the *request* argument determines the precise action of the call:

0 This request is the only one used by the child process; it declares that the process is to be traced by its parent. All the other arguments are ignored. Peculiar results will ensue if the parent does not expect to trace the child.

1,2 The word in the child process's address space at *addr* is returned. If I and D space are separated, request 1 indicates I space, 2 D space. *Addr* must be even. The child must be stopped. The input *data* is ignored.

3 The word of the system's per-process data area corresponding to *addr* is returned. *Addr* must be even and less than 512. This space contains the registers and other information about the process; its layout corresponds to the *user* structure in the system.

4,5 The given *data* is written at the word in the process's address space corresponding to *addr*, which must be even. No useful value is returned. If I and D space are separated, request 4 indicates I space, 5 D space. Attempts to write in pure procedure fail if another process is executing the same file.

6 The process's system data is written, as it is read with request 3. Only a few locations can be written in this way: the general registers, the floating point status and registers, and certain bits of the processor status word.

7 The *data* argument is taken as a signal number and the child's execution continues as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal which caused the stop should be ignored, or that value fetched out of the process's image indicating which signal caused the stop. Signal number –1 transmits no signal, but causes the process to continue, then stop as soon as possible after having executed at least one instruction; the signal number received by the parent at this stop is SIGTRC.

8 The traced process terminates.

As indicated, these calls (except for request 0) can be used only when the subject process has stopped. The *wait* call is used to determine when a process stops; in such a case the "termination" status returned by *wait* has the value 0177 to indicate stoppage rather than genuine termination.

To forestall possible fraud, *ptrace* inhibits the set-user-id facility on subsequent *exec*(2) calls. If a traced process calls *exec*, it will stop before executing the first instruction of the new image showing signal SIGTRC.

SEE ALSO

adb(1), signal(2), wait(2)

DIAGNOSTICS

The value −1 is returned if *request* is invalid, *pid* is not a traceable process, *addr* is out of bounds, or *data* specifies an illegal signal number.

BUGS

The request 0 call should be able to specify signals which are to be treated normally and not cause a stop. In this way, for example, programs with simulated floating point (which use "illegal instruction" signals at a very high rate) could be efficiently debugged.

The error indication, −1, is a legitimate function value; *errno* (see *intro*(2)) can be used to disambiguate.

It should be possible to stop a process on occurrence of a system call; in this way a completely controlled environment could be provided.

ASSEMBLER

(ptrace = 26.)
(data in r0)
sys ptrace; pid; addr; request
(value in r0)

NAME

read − read from file

SYNOPSIS

read (fildes, buffer, nbytes)
char *buffer;

DESCRIPTION

A file descriptor is a word returned from a successful *open*, *creat*, or *pipe* call. *Buffer* is the location of *nbytes* contiguous bytes into which the input will be placed. It is not guaranteed that all *nbytes* bytes will be read; for example if the file refers to a typewriter at most one line will be returned. In any event, the number of characters read is returned (in r0).

If the returned value is 0, then end-of-file has been reached.

SEE ALSO

open(2), dup(2), close(2), creat(2), pipe(2), write(2)

DIAGNOSTICS

As mentioned, 0 is returned when the end of the file has been reached. If the read was otherwise unsuccessful the error bit (c-bit) is set. Many conditions can generate an error: physical I/O errors, bad buffer address, preposterous *nbytes*, file descriptor not that of an input file, etc. From C, a −1 return indicates an error.

ASSEMBLER

(read = 3.)
(file descriptor in r0)
sys read; buffer; nbytes
(byte count in r0)

NAME

reboot − transfer control to DEC rom and reboot

SYNOPSIS

reboot (unit, name)
char *name;

DESCRIPTION

Reboot causes termination of the current system with control being given to the DEC YC rom. The *unit* argument is the value that would normally have been placed in the console switches before starting the rom program manually. For an RP06 disk drive unit 0, for example, the unit specified would be 070. The *name* argument is the name of the file to be passed to the boot routine which is loaded by the rom. This file name is stored in a place known to both the reboot system call and the boot routine. The file name is limited to 15 characters in length including a terminating newline which must be supplied.

This system call is limited to the super-user for obvious reasons.

DIAGNOSTICS

Reboot will not return if successful. C-bit is set on error with the error code in r0; from C, −1 return indicates failure.

SEE ALSO

reboot(1M)

ASSEMBLER

(syscb = 45.; reboot = 2.)
(reboot in R1)
(unit in R0)
sys syscb; name;

NAME

seek − move read/write pointer

SYNOPSIS

seek (fildes, offset, ptrname)

DESCRIPTION

Seek has been droped in this version of the library. Use *lseek*(2) instead.

ASSEMBLER

(seek = 19.)
(file descriptor in r0)
sys seek; offset; ptrname