INTRODUCTION TO SYSTEM CALLS

Section II of this manual lists all the entries into the system. 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 successful 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 II, since many errors are possible for most of the calls. Here is a list of the error numbers, their names inside the system (for the benefit of system-readers), and the messages available using *perror.* A short explanation is also provided.

0 – (unused)

1 EPERM Not owner and not super-user

Typically this error indicates an attempt to modify a file in some way forbidden except to its owner. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

2 ENOENT No such file or directory

This error occurs when a file name is specified and the file should exist but doesn’t, or when one of the directories in a path name does not exist.

3 ESRCH No such process

The process whose number was given to *signal* does not exist, or is already dead.

4 EINTR Interrupted system call

An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.

5 EIO I/O error

Some physical I/O error occurred during a *read* or *write.* This error may in some cases occur on a call following the one to which it actually applies.

6 ENXIO No such device or address

I/O on a special file refers to a subdevice which does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not dialed in or no disk pack is loaded on a drive.

7 E2BIG Arg list too long

An argument list longer than the maximum allowable (counting the null at the end of each argument) is presented to *exec.* The maximum is a configuration dependent parameter.

8 ENOEXEC Exec format error

A request is made to execute a file which, although it has the appropriate permissions, does not start with one of the magic numbers 407, 410, or 411.

9 EBADF Bad file number
Either a file descriptor refers to no open file, or a read (resp. write) request is made to a file which is open only for writing (resp. reading).

10 ECHILD No children
Wait and the process has no living or unwaited-for children.

11 EAGAIN No more processes
In a *fork,* the system's process table is full and no more processes can for the moment be created.

12 ENOMEM Not enough core
During an *exec* or *break,* a program asks for more core than the system is able to supply. This is not a temporary condition; the maximum core size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments is such as to require more than the existing 8 segmentation registers.

13 EACCES Permission denied
An attempt was made to access a file in a way forbidden by the protection system.

14 EFAULT Memory fault
A memory fault occurred while passing data between the user and the system. Most likely the result of bad arguments to the system call.

15 ENOTBLK Block device required
A plain file was mentioned where a block device was required, e.g., in *mount.*

16 EBUSY Mount device busy
An attempt to mount a device that was already mounted or an attempt was made to dismount a device on which there is an open file or some process's current directory.

17 EEXIST File exists
An existing file was mentioned in an inappropriate context, e.g., *link.*

18 EXDEV Cross-device link
A link to a file on another device was attempted.

19 ENODEV No such device
An attempt was made to apply an inappropriate system call to a device; e.g., read a write-only device.

20 ENOTDIR Not a directory
A non-directory was specified where a directory is required, for example in a path name or as an argument to *chdir.*

21 EISDIR Is a directory
An attempt to write on a directory.

22 EINVAL Invalid argument
Some invalid argument: currently, dismounting a non-mounted device, mentioning an unknown signal in *signal,* and giving an unknown request in *stty* to the TIU special file.

23 ENFILE File table overflow
The system’s table of open files is full, and temporarily no more *opens* can be accepted.

24 EMFILE Too many open files
Only 15 files can be open per process.

25 ENOTTY Not a terminal
The file mentioned in *stty* or *gtty* is not a terminal or one of the other devices to which these calls apply.

26 ETXTBSY Text file busy
An attempt to execute a pure-procedure program which is currently open for writing (or reading!). Also an attempt to open for writing a pure-procedure program that is being executed.

27 EFBIG File too large
An attempt to make a file larger than the maximum of 32768 blocks.

28 ENOSPC No space left on device
During a *write* to an ordinary file, there is no free space left on the device.

29 ESPIPE Seek on pipe
A *seek* was issued to a pipe. This error should also be issued for other non-seekable devices.

30 EROFS Read-only file system
An attempt to modify a file or directory was made on a device mounted read-only.

31 EMLINK Too many links
An attempt to make more than 127 links to a file.

32 EPIPE Write on broken pipe
A write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

NAME

access − determine accessibility of file

SYNOPSIS

(access = 33.)
sys access; name; mode

access(name, mode)
char *name;

DESCRIPTION

Access checks the given file *name* for accessibility according to *mode*, which is 4 (read), 2 (write) or 1 (execute) or a combination thereof. An appropriate error indication is returned if one or more of the desired access modes would not be granted.

The user and group IDs with respect to which permission is checked are the real UID and GID of the process, so this call is useful to set-UID programs.

Notice that it is only access bits that are checked. A directory may be announced as writable by*access*, b ut an attempt to open it for writing will fail (although files may be created there); a file may look excutable, but *exec*(II) will fail unless it is in proper format.

SEE ALSO

stat(II)

DIAGNOSTICS

C-bit is set on disallowed accesses, and the error code is in r0; from C, −1 is returned and the error code is in *errno*. 0 is returned from successful tests.

NAME

alarm − schedule signal after specified time

SYNOPSIS

(alarm = 27.)
(seconds in r0)
sys alarm

alarm(seconds)
int seconds;

DESCRIPTION

Alarm causes signal number 14 to be sent to the invoking process in a number of seconds given by the argument. Unless caught, the signal terminates the process.

Alarm requests are not stacked; successive calls reset the alarm clock. If the argument is 0, any alarm request is cancelled. Because the clock has a 1-second resolution, the signal may occur up to one second early; because of scheduling delays, resumption of execution of when the signal is caught may be delayed an arbitrary amount. The longest specifiable delay time is 65535 seconds. The old value of the alarm clock is returned in r0. In C, that value is returned.

SEE ALSO

pause(II), sleep(III)

NAME

break, brk, sbrk − change core allocation

SYNOPSIS

(break = 17.)
sys break; addr

char *brk(addr)

char *sbrk(incr)

DESCRIPTION

Break sets the system’s idea of the lowest location not used by the program (called the break) to *addr* (rounded up to the next multiple of 64 bytes). Locations not less than *addr* and below the stack pointer are not in the address space and will thus cause a memory violation if accessed.

From C, *brk* will set the break to *addr.* The old break is returned.

In the alternate entry *sbrk, incr* more bytes are added to the program’s data space and a pointer to the start of the new area is returned.

When a program begins execution via *exec* the break is set at the highest location defined by the program and data storage areas. Ordinarily, therefore, only programs with growing data areas need to use *break.*

SEE ALSO

exec(II), alloc(III), end(III)

DIAGNOSTICS

The c-bit is set if the program requests more memory than the system limit or if more than 8 segmentation registers would be required to implement the break. From C, −1 is returned for these errors.

BUGS

Setting the break in the range 0177700 to 0177777 is the same as setting it to zero.

NAME

chdir − change working directory

SYNOPSIS

(chdir = 12.)
sys chdir; dirname

chdir(dirname)
char *dirname;

DESCRIPTION

Dirname is the address of the pathname of a directory, terminated by a null byte. *Chdir* causes this directory to become the current working directory.

SEE ALSO

chdir(I)

DIAGNOSTICS

The error bit (c-bit) is set if the given name is not that of a directory or is not readable. From C, a −1 returned value indicates an error, 0 indicates success.

NAME

chmod − change mode of file

SYNOPSIS

(chmod = 15.)
sys chmod; name; mode

chmod(name, mode)
char *name;

DESCRIPTION

The file whose name is given as the null-terminated string pointed to by *name* has its mode changed to *mode.* Modes are constructed by ORing together some combination of the following:

4000 set user ID on execution
2000 set group ID on execution
1000 save text image after execution
0400 read by owner
0200 write by owner
0100 execute (search on directory) by owner
0070 read, write, execute (search) by group
0007 read, write, execute (search) by others

Only the owner of a file (or the super-user) may change the mode. Only the super-user can set the 1000 mode.

SEE ALSO

chmod(I)

DIAGNOSTIC

Error bit (c-bit) set if *name* cannot be found or if current user is neither the owner of the file nor the super-user. From C, a −1 returned value indicates an error, 0 indicates success.

NAME

chown − change owner and group of a file

SYNOPSIS

(chown = 16.)
sys chown; name; owner

chown(name, owner)
char *name;

DESCRIPTION

The file whose name is given by the null-terminated string pointed to by *name* has its owner and group changed to the low and high bytes of *owner* respectively.

SEE ALSO

chown(I), chgrp(I), passwd(V)

DIAGNOSTICS

The error bit (c-bit) is set on illegal owner changes. From C, a −1 returned value indicates error, 0 indicates success.

NAME

close − close a file

SYNOPSIS

(close = 6.)
(file descriptor in r0)
sys close

close(fildes)

DESCRIPTION

Given a file descriptor such as returned from an *open*, *creat*, or *pipe* call, *close* closes the associated file. A close of all files is automatic on *exit*, but since processes are limited to 15 simultaneously open files, *close* is necessary for programs which deal with many files.

SEE ALSO

creat(II), open(II), pipe(II)

DIAGNOSTICS

The error bit (c-bit) is set for an unknown file descriptor. From C, a −1 indicates an error, 0 indicates success.

NAME

creat − create a new file

SYNOPSIS

(creat = 8.)
sys creat; name; mode
(file descriptor in r0)

creat(name, mode)
char *name;

DESCRIPTION

Creat creates a new file or prepares to rewrite an existing file called *name*, given as the address of a null-terminated string. If the file did not exist, it is given mode *mode.* See *chmod(II)* for the construction of the *mode* argument.

If the file did exist, its mode and owner remain unchanged but it is truncated to 0 length.

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

The *mode* given is arbitrary; it need not allow writing. This feature is used by programs which deal with temporary files of fixed names. The creation is done with a mode that forbids writing. Then if a second instance of the program attempts a *creat,* an error is returned and the program knows that the name is unusable for the moment.

SEE ALSO

write(II), close(II), stat(II)

DIAGNOSTICS

The error bit (c-bit) may be set 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; the file is a directory; there are already too many files open.

From C, a −1 return indicates an error.

NAME

csw − read console switches

SYNOPSIS

(csw = 38.)
sys csw

getcsw( )

DESCRIPTION

The setting of the console switches is returned (in r0).

NAME

dup − duplicate an open file descriptor

SYNOPSIS

(dup = 41.)
(file descriptor in r0)
sys dup

dup(fildes)
int fildes;

DESCRIPTION

Given a file descriptor returned from an *open*, *pipe*, or *creat* call, *dup* will allocate another file descriptor synonymous with the original. The new file descriptor is returned in r0.

Dup is used more to reassign the value of file descriptors than to genuinely duplicate a file descriptor. Since the algorithm to allocate file descriptors returns the lowest available value, combinations of *dup* and *close* can be used to manipulate file descriptors in a general way. This is handy for manipulating standard input and/or standard output.

SEE ALSO

creat(II), open(II), close(II), pipe(II)

DIAGNOSTICS

The error bit (c-bit) is set if: the given file descriptor is invalid; there are already too many open files. From C, a −1 returned value indicates an error.

NAME

exec, execl, execv − execute a file

SYNOPSIS

(exec = 11.)
sys exec; name; args
name: <...\0>
args: arg0; arg1; ...; 0
arg0: <...\0>
arg1: <...\0>
...

execl(name, arg0, arg1, ..., argn, 0)
char *name, *arg0, *arg1, ..., *argn;

execv(name, argv)
char *name;
char *argv[ ];

DESCRIPTION

Exec overlays the calling process with the named file, then transfers to the beginning of the core image of the file. There can be no return from the file; the calling core image is lost.

Files remain open across *exec* calls. Ignored signals remain ignored across *exec*, 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. *Exec* 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 *exec* is a pointer to the name of the file to be executed. The second is the address of a null-terminated list of pointers to arguments to be passed to the file. Conventionally, the first argument is the name of the file. Each pointer addresses a string terminated by a null byte.

Once 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
        arg0
        ...
        argn
        −1

arg0:   <arg0\0>
        ...
argn:   <argn\0>
```

From C, two interfaces are available. *execl* is useful when a known file with known arguments is being called; the arguments to *execl* 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 *execv* version is useful when the number of arguments is unknown in advance; the arguments to *execv* 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 *execv*, since *argv[argc]* is −1 and not 0.

SEE ALSO

fork(II), pexec(III)

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, or if the arguments require more than 5120 bytes a return from *exec* constitutes the diagnostic; the error bit (c-bit) is set. Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed. From C the returned value is −1.

BUGS

Only 5120 characters of arguments are allowed.

NAME

exit − terminate process

SYNOPSIS

(exit = 1.)
(status in r0)
sys exit

exit(status)
int status;

DESCRIPTION

Exit is the normal means of terminating a process. *Exit* closes all open files of the process, and notifies the parent process, if it is executing a *wait.* The low byte of r0 (resp. the argument to *exit*) is available as status to the parent process.

This call can never return.

SEE ALSO

wait(II)

NAME

fork − spawn new process

SYNOPSIS

(fork = 2.)
sys fork
(new process return)
(old process return)

fork( )

DESCRIPTION

Fork is the only way new processes are created. The new process's core image is a copy of that of the caller of *fork.* The only distinction is the return location and the fact that r0 in the old (parent) process contains the process ID of the new (child) process. This process ID is used by *wait.*

The two returning processes share all open files that existed before the call. In particular, this is the way that standard input and output files are passed and also how pipes are set up.

From C, the child process receives a 0 return, and the parent receives a non-zero number which is the process ID of the child; a return of −1 indicates inability to create a new process.

SEE ALSO

wait(II), exec(II), pexec(III)

DIAGNOSTICS

The error bit (c-bit) is set in the old process if a new process could not be created because of lack of process space. From C, a return of −1 indicates an error.

NAME

fstat − get status of open file

SYNOPSIS

(fstat = 28.)
(file descriptor in r0)
sys fstat; buf

fstat(fildes, buf)
struct inode *buf;

DESCRIPTION

This call is identical to *stat*, except that it operates on open files instead of files given by name. It is most often used to get the status of the standard input and output files, whose names are unknown.

SEE ALSO

stat(II)

DIAGNOSTICS

The error bit (c-bit) is set if the file descriptor is unknown; from C, a −1 return indicates an error, 0 indicates success.

NAME

getgid − get group identifications

SYNOPSIS

(getgid = 47.)
sys getgid

getgid( )

DESCRIPTION

Getgid returns a word (in r0), the low byte of which contains the real group ID of the current process. The high byte contains the effective group ID of the current process. The real group ID identifies the group of the person who is logged in, in contradistinction to the effective group ID, which determines his access permission at the moment. It is thus useful to programs which operate using the ‘‘set group ID’’ mode, to find out who invoked them.

SEE ALSO

setgid(II)

NAME

getpid − get process identification

SYNOPSIS

(getpid = 20.)
sys getpid
(pid in r0)

getpid( )

DESCRIPTION

Getpid returns the process ID of the current process. Most often it is used to generate uniquely-named temporary files.

NAME

getuid − get user identifications

SYNOPSIS

(getuid = 24.)
sys getuid

getuid( )

DESCRIPTION

Getuid returns a word (in r0), the low byte of which contains the real user ID of the current process. The high byte contains the effective user ID of the current process. The real user ID identifies the person who is logged in, in contradistinction to the effective user ID, which determines his access permission at the moment. It is thus useful to programs which operate using the ‘‘set user ID’’ mode, to find out who invoked them.

SEE ALSO

setuid(II)

NAME

gtty − get terminal status

SYNOPSIS

(gtty = 32.)
(file descriptor in r0)
sys gtty; arg
arg: .=.+6

gtty(fildes, arg)
int arg[3];

DESCRIPTION

Gtty stores in the three words addressed by *arg* the status of the terminal whose file descriptor is given in r0 (resp. given as the first argument). The format is the same as that passed by*stty* .

SEE ALSO

stty(II)

DIAGNOSTICS

Error bit (c-bit) is set if the file descriptor does not refer to a terminal. From C, a −1 value is returned for an error, 0, for a successful call.

NAME

indir − indirect system call

SYNOPSIS

(indir = 0.)
sys indir; syscall

DESCRIPTION

The system call at the location *syscall* is executed. Execution resumes after the *indir* call.

The main purpose of *indir* is to allow a program to store arguments in system calls and execute them out of line in the data segment. This preserves the purity of the text segment.

If *indir* is executed indirectly, it is a no-op. If the instruction at the indirect location is not a system call, the executing process will get a fault.

NAME

kill − send signal to a process

SYNOPSIS

(kill = 37.)
(process number in r0)
sys kill; sig

kill(pid, sig);

DESCRIPTION

Kill sends the signal *sig* to the process specified by the process number in r0. See *signal*(II) for a list of signals.

The sending and receiving processes must have the same effective user ID. The super-user can kill any process.

If the process number is 0, the signal is sent to all processes which have the same process group number as the sender. If the process number is less than 0, the signal is sent to all processes for which the sender has permission. In both of the above cases, process 0 and process 1 are excluded. Note, process 0 is really the scheduler, and process numbers must be positive to avoid confusion with error indications in C.

SEE ALSO

kill(I), signal(II)

DIAGNOSTICS

The error bit (c-bit) is set if the process does not have permission, or if the process does not exist. From C, a −1 return indicates an error.

NAME

link − link to a file

SYNOPSIS

(link = 9.)
sys link; name1; name2

link(name1, name2)
char *name1, *name2;

DESCRIPTION

A link to *name1* is created; the link has the name *name2.* Either name may be an arbitrary path name.

SEE ALSO

ln(I), unlink(II)

DIAGNOSTICS

The error bit (c-bit) is set when *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; when an attempt is made to link to a file on another file system; when more than 127 links are made. From C, a −1 return indicates an error, a 0 return indicates success.

NAME

logname, logdir, logtty, logpost − login information

SYNOPSIS

**char	*logname(), *logdir(), *logtty();**
logpost(buf)
**char	*buf;**

DESCRIPTION

Logname returns a pointer to the null-terminated login name (fits in 8 characters).

Logdir returns a pointer to the null-terminated login directory pathname (fits in 22 char).

Logtty returns a pointer to the tty letter.

These data are created by *login*(I) using the function *logpost* which is executable only by the super-user.

This function is kept in the **−lPW** library.

SEE ALSO

login(I), udata(II)

DIAGNOSTICS

Same as for *udata*(II).

NAME

mknod − make a directory or a special file

SYNOPSIS

(mknod = 14.)
sys mknod; name; mode; addr

mknod(name, mode, addr)
char *name;

DESCRIPTION

Mknod creates a new file whose name is the null-terminated string pointed to by *name.* The mode of the new file (including directory and special file bits) is initialized from *mode.* The first physical address of the file is initialized from *addr.* Note that in the case of a directory, *addr* should be zero. In the case of a special file, *addr* specifies which special file.

Mknod may be invoked only by the super-user.

SEE ALSO

mkdir(I), mknod(VIII), fs(V)

DIAGNOSTICS

Error bit (c-bit) is set if the file already exists or if the user is not the super-user. From C, a −1 value indicates an error.

NAME

mount − mount file system

SYNOPSIS

(mount = 21.)
sys mount; special; name; rwflag

mount(special, name, rwflag)
char *special, *name;

DESCRIPTION

Mount announces to the system that a removable file system has been mounted on the block-structured special file *special;* from now on, references to file *name* will refer to the root file on the newly mounted file system. *Special* and *name* are pointers to null-terminated strings containing the appropriate path names.

Name must exist already. Its old contents are inaccessible while the file system is mounted.

The *rwflag* argument determines whether the file system can be written on; if it is 0 writing is allowed, if non-zero no writing is done. Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

Only the super-user can execute mount.

SEE ALSO

mount(VIII), umount(II)

DIAGNOSTICS

Error bit (c-bit) set if: *special* is inaccessible or not an appropriate file; *name* does not exist; *special* is already mounted; *name* is in use; there are already too many file systems mounted.

NAME

nice − set program priority

SYNOPSIS

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

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 4 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.

The actual running priority of a process is the *priority* argument plus a number that ranges from 100 to 127 depending on the cpu usage of the process.

SEE ALSO

nice(I)

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.

NAME

open − open for reading or writing

SYNOPSIS

(open = 5.)
sys open; name; mode
(file descriptor in r0)

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,* and *close.*

SEE ALSO

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

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 too many files are open. From C, a −1 value is returned on an error.

NAME

pause − indefinite wait

SYNOPSIS

(pause = 29.)
sys pause

pause ();

DESCRIPTION

Pause causes its caller to suspend execution indefinitely. A caught signal is processed normally. The most plausible use of *pause* is in conjunction with an alarm-clock signal: *alarm(II).*

SEE ALSO

alarm(II), signal(II)

NAME

pipe − create an interprocess channel

SYNOPSIS

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

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 generate a fatal signal (*signal*(II)); if the signal is ignored, an error is returned on the write.

SEE ALSO

sh(I), read(II), write(II), fork(II)

DIAGNOSTICS

The error bit (c-bit) is set if too many files are already open. From C, a −1 returned value indicates an error. A signal is generated if a write on a pipe with only one end is attempted.

NAME

profil − execution time profile

SYNOPSIS

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

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: 177777(8) gives a 1-1 mapping of pc's to words in *buff;* 77777(8) maps each pair of instruction words together. 2(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 also turned off when an *exec* is executed but remains on in child and parent both after a *fork.*

SEE ALSO

monitor(III), prof(I)

NAME

ptrace − process trace

SYNOPSIS

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

ptrace(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*(II) for the list. Then the traced process enters a stopped state and its parent is notified via *wait(II).* 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 (in r0). Request 1 indicates the instruction space, 2 indicates the data space. *addr* must be e ven. 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 in the data area. 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. Request 4 specifies instruction space, 5 specifies data space. Attempts to write in pure procedure result in termination of the child, instead of going through or causing an error for the parent.

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.

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(II)* calls.

SEE ALSO

wait(II), signal(II), cdb(I)

DIAGNOSTICS

From assembler, the c-bit (error bit) is set on errors; from C, −1 is returned and *errno* has the error code.

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.

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

NAME

read − read from file

SYNOPSIS

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

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

DESCRIPTION

A file descriptor is a word returned from a successful *open*, *creat*, *dup,* 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 terminal 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(II), creat(II), dup(II), pipe(II)

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. From C, a −1 return indicates the error.

NAME

seek − move read/write pointer

SYNOPSIS

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

seek(fildes, offset, ptrname)

DESCRIPTION

The file descriptor refers to a file open for reading or writing. The read (resp. write) pointer for the file is set as follows:

if *ptrname* is 0, the pointer is set to *offset.*

if *ptrname* is 1, the pointer is set to its current location plus *offset.*

if *ptrname* is 2, the pointer is set to the size of the file plus *offset.*

if *ptrname* is 3, 4 or 5, the meaning is as above for 0, 1 and 2 except that the offset is multiplied by 512.

If *ptrname* is 0 or 3, *offset* is unsigned, otherwise it is signed.

SEE ALSO

open(II), creat(II), tell(II)

DIAGNOSTICS

The error bit (c-bit) is set for an undefined file descriptor. From C, a −1 return indicates an error.

NAME

setgid − set process group ID

SYNOPSIS

(setgid = 46.)
(group ID in r0)
sys setgid

setgid(gid)

DESCRIPTION

The group ID of the current process is set to the argument. Both the effective and the real group ID are set. This call is only permitted to the super-user or if the argument is the real group ID.

SEE ALSO

getgid(II)

DIAGNOSTICS

Error bit (c-bit) is set as indicated; from C, a −1 value is returned.

NAME

setpgrp − set process group number

SYNOPSIS

(setpgrp = 39.; not in assembler)
sys setpgrp

setpgrp( )

DESCRIPTION

Setpgrp sets the process group number of the process to the process ID of the process. The process ID is guaranteed to be unique among the current process IDs and process group numbers, so that the new process group number will be unique. Process group numbers are used to group processes for catching signals.

SEE ALSO

kill(II), signal(II)

NAME

setuid − set process user ID

SYNOPSIS

(setuid = 23.)
(user ID in r0)
sys setuid

setuid(uid)

DESCRIPTION

The user ID of the current process is set to the argument. Both the effective and the real user ID are set. This call is only permitted to the super-user or if the argument is the real user ID.

SEE ALSO

getuid(II)

DIAGNOSTICS

Error bit (c-bit) is set as indicated; from C, a −1 value is returned.

NAME

signal − catch or ignore signals

SYNOPSIS

(signal = 48.)
sys signal; sig; label
(old value in r0)

signal(sig, func)
int (*func)( );

DESCRIPTION

A *signal* is generated by some abnormal event, initiated either by user at a terminal (quit, interrupt), by a program error (bus error, etc.), or by request of another program (kill). Normally all signals cause termination of the receiving process, but this call allows them either to be ignored or to cause an interrupt to a specified location. Here is the list of signals:

1	hangup
2	interrupt
3*	quit
4*	illegal instruction (not reset when caught)
5*	trace trap (not reset when caught)
6*	IOT instruction
7*	EMT instruction
8*	floating point exception
9	kill (cannot be caught or ignored)
10*	bus error
11*	segmentation violation
12*	nonexistent system call
13	write on a pipe with no one to read it
14	alarm clock
15	software termination (catchable kill)

The starred signals in the list above cause a core image if not caught or ignored.

In the assembler call, if *label* is 0, the process is terminated when the signal occurs; this is the default action. If *label* is odd, the signal is ignored. Any other even *label* specifies an address in the process where an interrupt is simulated. An RTI or RTT instruction will return from the interrupt. Except as indicated, a signal is reset to 0 after being caught. Thus if it is desired to catch every such signal, the catching routine must issue another *signal* call.

In C, if *func* is 0, the default action for signal *sig* (termination) is reinstated. If *func* is 1, the signal is ignored. If *func* is non-zero and even, it is assumed to be the address of a function entry point. When the signal occurs, the function will be called. A return from the function will continue the process at the point it was interrupted. As in the assembler call, *signal* must in general be called again to catch subsequent signals.

When a caught signal occurs during certain system calls, the call terminates prematurely. In particular, this can occur during a *read* or *write* on a slow device (like a terminal, but not a disk file), and during *wait.* When such a signal occurs, the saved user status is arranged in such a way that when return from the signal-catching takes place, it will appear that the system call returned a characteristic error status. The user’s program may then, if it wishes, re-execute the call.

The value of the call is the old action defined for the signal.

After a *fork(II)* the child inherits all signals. *Exec(II)* resets all caught signals to default action.

SEE ALSO

kill(I), kill(II), ptrace(II), reset(III)

DIAGNOSTICS

The error bit (c-bit) is set if the given signal is out of range. In C, a −1 indicates an error.

NAME

stat − get file status

SYNOPSIS

(stat = 18.)
sys stat; name; buf

stat(name, buf)
char *name;
struct inode *buf;

DESCRIPTION

Name points to a null-terminated string naming a file; *buf* is the address of a 36(10) byte buffer into which information is placed concerning the file. It is unnecessary to have any permissions at all with respect to the file, but all directories leading to the file must be readable. After *stat*, *buf* has the following structure (starting offset given in bytes):

```
struct inode {
        char    minor;          /* +0: minor device of i-node */
        char    major;          /* +1: major device */
        int     inumber;        /* +2 */
        int     flags;          /* +4: see below */
        char    nlinks;         /* +6: number of links to file */
        char    uid;            /* +7: user ID of owner */
        char    gid;            /* +8: group ID of owner */
        char    size0;          /* +9: high byte of 24-bit size */
        int     size1;          /* +10: low word of 24-bit size */
        int     addr[8];        /* +12: block numbers or device number */
        int     actime[2];      /* +28: time of last access */
        int     modtime[2];     /* +32: time of last modification */
};
```

The flags are as follows:

```
 100000   i-node is allocated
 060000   2-bit file type:
        000000   plain file
        040000   directory
        020000   character-type special file
        060000   block-type special file.
 010000   large file
 004000   set user-ID on execution
 002000   set group-ID on execution
 001000   save text image after execution
 000400   read (owner)
 000200   write (owner)
 000100   execute (owner)
 000070   read, write, execute (group)
 000007   read, write, execute (others)
```

SEE ALSO

ls(I), fstat(II), fs(V)

DIAGNOSTICS

Error bit (c-bit) is set if the file cannot be found. From C, a −1 return indicates an error.

NAME

stime − set time

SYNOPSIS

(stime = 25.)
(time in r0-r1)
sys stime

stime(tbuf)
int tbuf[2];

DESCRIPTION

Stime sets the system's idea of the time and date. Time is measured in seconds from 0000 GMT Jan 1, 1970. Only the super-user may use this call.

SEE ALSO

date(I), time(II), ctime(III)

DIAGNOSTICS

Error bit (c-bit) set if user is not the super-user.

NAME

stty − set mode of terminal

SYNOPSIS

(stty = 31.)
(file descriptor in r0)
sys stty; arg
arg: .byte ispeed, ospeed; .byte erase, kill; mode

```
stty(fildes, arg)
struct {
        char    ispeed, ospeed;
        char    erase, kill;
        int     mode;
} *arg;
```

DESCRIPTION

Stty sets mode bits and character speeds for the terminal whose file descriptor is passed in r0 (resp. is the first argument to the call). First, the system delays until the terminal is quiescent. The input and output speeds are set from the first two bytes of the argument structure as indicated by the following table, which corresponds to the speeds supported by the DH-11 interface.

0	(hang up modem)
1	50 baud
2	75 baud
3	110 baud
4	134.5 baud
5	150 baud
6	200 baud
7	300 baud
8	600 baud
9	1200 baud
10	1800 baud
11	2400 baud
12	4800 baud
13	9600 baud
14	External A
15	External B

In the current configuration, only 110, 150 and 300 baud are really supported on dial-up lines. The half-duplex line discipline required for the 202 modem (1200 baud) is not supplied.

The next two characters of the argument structure specify the erase and kill characters respectively. (Defaults are # and @.)

The *mode* contains several bits that determine the system's treatment of the terminal:

100000	select one of two types of backspace delays
040000	select one of two types of form-feed and vertical-tab delays
030000	select one of four types of carriage-return delays
006000	select one of four types of tab delays
001400	select one of four types of new-line delays
000200	even parity allowed on input
000100	odd parity allowed on input
000040	raw mode

000020	map CR into LF; echo LF or CR as CR-LF
000010	echo (full duplex)
000004	map upper case to lower on input
000002	echo and print tabs as spaces
000001	hang up (drop ‘data terminal ready’) after last close

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 indicates no delay.

Backspace delays are currently unimplemented.

Form-feed/vertical-tab delay type 1 lasts about 2 seconds.

Carriage-return delay types 1 and 2 last about .09 seconds, and type 3 lasts about .15 seconds. Types 2 and 3 have the side effect of not transmitting a carriage-return if at the leftmost column.

New-line delay type 1 is dependent on the current column and is tuned for the TELETYPE® Model 37. Type 2 lasts about .03 seconds and type 3 lasts about .15 seconds.

Tab delay type 1 is dependent on the amount of movement and is tuned for the TELETYPE Model 37. Other types are unimplemented and are 0.

Characters with the wrong parity, as determined by bits 0200 and 0100, are ignored.

In raw mode, every character is passed immediately to the program without waiting until a full line has been typed. No erase or kill processing is done; the end-of-file character (EOT), the interrupt character (DEL) and the quit character (FS) are not treated specially.

Mode 020 causes input carriage returns to be turned into new-lines; input of either CR or LF causes LF-CR both to be echoed (used for terminals without the newline function, i.e. most).

The upper case mode is used on terminals without lower case, see *tty(IV).*

The hangup mode 01 causes the line to be disconnected when the last process with the line open closes it or terminates. It is useful when a port is to be used for some special purpose; for example, if it is associated with an ACU used to place outgoing calls.

This system call is also used with certain special files other than terminals, and is system dependent.

SEE ALSO

stty(I), gtty(II), tty(IV)

DIAGNOSTICS

The error bit (c-bit) is set if the file descriptor does not refer to a terminal. From C, a negative value indicates an error.

NAME

sync − update super-block

SYNOPSIS

(sync = 36.)
sys sync

DESCRIPTION

Sync causes all information in core memory that should be on disk 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 *icheck, df,* etc. It is mandatory before a boot.

SEE ALSO

sync(I)

NAME

tell − get file offset

SYNOPSIS

(tell = 40.)
(file descriptor in r0)
sys tell
(offset in r0-r1)

```
long    tell(file)
int file;
```

DESCRIPTION

Tell returns the current read/write pointer associated with the open file whose descriptor is specified as argument.

SEE ALSO

seek (II)

DIAGNOSTICS

C-bit set or −1 returned for an unknown file descriptor.

NAME

time − get date and time

SYNOPSIS

(time = 13.)
sys time

time(tvec)
long *tvec;

DESCRIPTION

Time returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds. From *as*, the high order word is in the r0 register and the low order is in r1. From C, the user-supplied vector is filled in.

SEE ALSO

date(I), stime(II), ctime(III)

NAME

times − get process times

SYNOPSIS

(times = 43.)
sys times; buffer

times(buffer)
struct tbuffer *buffer;

DESCRIPTION

Times returns time-accounting information for the current process and for the terminated child processes of the current process. All times are in 1/60 seconds.

After the call, the buffer will appear as follows:

```
struct tbuffer {
        long    proc_user_time;
        long    proc_system_time;
        long    child_user_time;
        long    child_system_time;
};
```

The children times are the sum of the children's process times and their children's times.

SEE ALSO

time(I)

NAME

udata − get per-user data

SYNOPSIS

(pwbsys = 57.; udata = 1)
(pointer to buffer in r0)
(function in r1)
sys pwbsys; udata

DESCRIPTION

Udata is used to access a 32 byte section of the per-user process data region. If the*function* is zero, the section is read into the buffer given by the pointer. If the function is non-zero, the section is written from the buffer if super-user. The structure of the section is left to the user.

SEE ALSO

loginfo(I), loginfo(II)

DIAGNOSTICS

The error bit(c-bit) is set if the buffer can not be read or written, or if not super-user for write. From C, a −1 return indicates an error.

NAME

umount − dismount file system

SYNOPSIS

(umount = 22.)
sys umount; special

DESCRIPTION

Umount announces to the system that special file *special* is no longer to contain a removable file system. The file associated with the special file reverts to its ordinary interpretation; see *mount(II)*.

Only the super-user can execute umount.

SEE ALSO

umount(VIII), mount(II)

DIAGNOSTICS

Error bit (c-bit) set if no file system was mounted on the special file or if there are still active files on the mounted file system.

NAME

uname − get name of current PWB/UNIX

SYNOPSIS

(pwbsys = 57.; uname = 0)
(pointer to name in r0)
sys pwbsys; uname

uname(name)
char *name;

DESCRIPTION

Uname returns in *name* the 8 byte character name of the current PWB/UNIX. The name is not null-terminated. By convention, the name is of the form pwb?date. For example, pwba0401 would indicate that this is PWB/UNIX System A and that its operating system was last modified on April 1.

This function is kept in the **−lPW** library.

SEE ALSO

uname(I)

DIAGNOSTICS

The error bit(c-bit) is set if *name* can not be written. From C, a −1 return indicates an error.

NAME

unlink − remove directory entry

SYNOPSIS

(unlink = 10.)
sys unlink; name

unlink(name)
char *name;

DESCRIPTION

Name points to a null-terminated string. *Unlink* removes the entry for the file pointed to by *name* from its directory. If this entry was the last link to the file, the contents 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)

DIAGNOSTICS

The error bit (c-bit) is set to indicate that the file does not exist or that its directory cannot be written. Write permission is not required on the file itself. It is also illegal to unlink a directory (except for the super-user). From C, a −1 return indicates an error.

NAME

ustat − get file system statistics

SYNOPSIS

```
(pwbsys = 57.; ustat = 2)
(pointer to buf in r0)
(device number in r1)
sys     pwbsys; ustat
```

```
ustat(device, buf)
char    *buf;
```

DESCRIPTION

Ustat is designed to return a section of the super block of the mounted file system specified by *device. Device* is *addr[0]* of the inode of the mounted block-type special file. The structure of *buf* is:

```
struct {
        int     s_tfree;        /* total free */
        int     s_tinode;       /* total inodes free */
        char    s_fname[6];     /* filsys name */
        char    s_fpack[6];     /* filsys pack name */
}
```

This function is kept in the **−lPW** library.

SEE ALSO

fs(V)

DIAGNOSTICS

The error bit(c-bit) is set if *device* is not mounted or *buf* can not be written. From C, a −1 return indicates an error.

NAME

utime − update times in file

SYNOPSIS

(pwbsys = 57.; utime = 3)
(pointer to times in r0)
(pointer to name in r1)
sys pwbsys; utime

utime(name, times)
char *name, *times;

DESCRIPTION

Utime is used to set both the access and modification times of a file. Only the super-user may use this call. *Name* points to a null-terminated string naming a file, and *times* points to a structure containing two long integer time values:

```
struct {
        long int actime;    /* access time */
        long int modtime;/* modification time */
};
```

This function is kept in the **−lPW** library.

SEE ALSO

stat(II)

DIAGNOSTICS

The error bit(c-bit) is set if *name* does not exist, if not super-user, or if a read-only file system. From C, a −1 return indicates an error.

NAME

wait − wait for process to terminate

SYNOPSIS

(wait = 7.)
sys wait
(process ID in r0)
(status in r1)

wait(status)
int *status;

DESCRIPTION

Wait causes its caller to delay until one of its child processes terminates. If any child has died since the last *wait,* return is immediate; if there are no children, return is immediate with the error bit set (resp. with a value of −1 returned). The normal return yields the process ID of the terminated child (in r0). In the case of several children several *wait* calls are needed to learn of all the deaths.

If no error is indicated on return, the r1 high byte (resp. the high byte stored into *status*) contains the low byte of the child process r0 (resp. the argument of *exit*) when it terminated. The r1 (resp. *status*) low byte contains the termination status of the process. See *signal*(II) for a list of termination statuses (signals); 0 status indicates normal termination. A special status (0177) is returned for a stopped process which has not terminated and can be restarted. See *ptrace*(II). If the 0200 bit of the termination status is set, a core image of the process was produced by the system.

If the parent process terminates without waiting on its children, the initialization process (process ID = 1) inherits the children.

SEE ALSO

exit(II), fork(II), signal(II)

DIAGNOSTICS

The error bit (c-bit) is set if there are no children not previously waited for. From C, a returned value of −1 indicates an error.

NAME

write − write on a file

SYNOPSIS

(write = 4.)
(file descriptor in r0)
sys write; buffer; nbytes

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

DESCRIPTION

A file descriptor is a word returned from a successful *open, creat, dup,* or *pipe* call.

Buffer is the address of *nbytes* contiguous bytes which are written on the output file. The number of characters actually written is returned (in r0). It should be regarded as an error if this is not the same as requested.

Writes which are multiples of 512 characters long and begin on a 512-byte boundary in the file are more efficient than any others.

SEE ALSO

creat(II), open(II), pipe(II)

DIAGNOSTICS

The error bit (c-bit) is set on an error: bad descriptor, buffer address, or count; physical I/O errors. From C, a returned value of −1 indicates an error.