

NAME

intro, errno — introduction to system calls and error numbers

SYNOPSIS

```
#include <errno.h>
```

DESCRIPTION

Section 2 of this manual lists all the entries into the system. Most of these calls have an error return. An error condition is indicated by an otherwise impossible returned value. Almost always this is `-1`; the individual sections specify the details. An error number is also made available in the external variable `errno`. `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(3)`. The possible error numbers are not recited with each writeup in section 2, since many errors are possible for most of the calls. Here is a list of the error numbers, their names as defined in `<errno.h>`, and the messages available using `perror`.

0 Error 0
Unused.

1 EPERM Not owner

Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or super-user. 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` and `ptrace` 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 dialled in or no disk pack is loaded on a drive.

7 E2BIG Arg list too long

An argument list longer than 5120 bytes is presented to `exec`.

8 ENOEXEC Exec format error

A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number, see `a.out(5)`.

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 or the user is not allowed to create any more processes.
- 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 requires too many segmentation registers.
- 13 EACCES Permission denied
An attempt was made to access a file in a way forbidden by the protection system.
- 14 EFAULT Bad address
The system encountered a hardware fault in attempting to access the arguments of a 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 active file directory. (open file, current directory, mounted-on file, active text segment).
- 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: dismounting a non-mounted device, mentioning an unknown signal in *signal*, reading or writing a file for which *seek* has generated a negative pointer. Also set by math functions, see *intro(3)*.
- 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
Customary configuration limit is 20 per process.
- 25 ENOTTY Not a typewriter
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
The size of a file exceeded the maximum (about 10^9 bytes).
- 28 ENOSPC No space left on device
During a *write* to an ordinary file, there is no free space left on the device.
- 29 EPIPE Illegal seek
An *lseek* 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 32767 links to a file.
- 32 EPIPE 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.
- 33 EDOM Math argument
The argument of a function in the math package (3M) is out of the domain of the function.
- 34 ERANGE Result too large
The value of a function in the math package (3M) is unrepresentable within machine precision.

SEE ALSO

intro(3)

ASSEMBLER (PDP-11)

as /usr/include/sys.s file ...

The PDP11 assembly language interface is given for each system call. The assembler symbols are defined in `'/usr/include/sys.s'`.

Return values appear in registers `r0` and `r1`; it is unwise to count on these registers being preserved when no value is expected. An erroneous call is always indicated by turning on the `c-bit` of the condition codes. The error number is returned in `r0`. 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.

On the Interdata 8/32, the system call arguments correspond well to the arguments of the C routines. The sequence is:

```
la    %2,errno
l      %0,&callno
svc    0,args
```

Thus register 2 points to a word into which the error number will be stored as needed; it is cleared if no error occurs. Register 0 contains the system call number; the nomenclature is identical to that on the PDP11. The argument of the *svc* is the address of the arguments, laid out in storage as in the C calling sequence. The return value is in register 2 (possibly 3 also, as in *pipe*) and is `-1` in case of error. The overflow bit in the program status word is also set when errors occur.

Under on a VAX-11, a system call follows exactly the same conventions as a C procedure. Namely, register **ap** points to a long word containing the number of arguments, and the arguments follow in successive long words. Values are returned in registers **r0** and **r1**. An error is indicated by setting the C (carry) bit in the processor status word; the error number is placed in **r0**.

NAME

access — determine accessibility of file

SYNOPSIS

```
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. Specifying mode 0 tests whether the directories leading to the file can be searched and the file exists.

An appropriate error indication is returned if *name* cannot be found or if any of the desired access modes would not be granted. On disallowed accesses -1 is returned and the error code is in *errno*. 0 is returned from successful tests.

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*, but an attempt to open it for writing will fail (although files may be created there); a file may look executable, but *exec* will fail unless it is in proper format.

SEE ALSO

stat(2)

ASSEMBLER (PDP-11)

(access = 33.)

sys access; name; mode

NAME

acct — turn accounting on or off

SYNOPSIS

acct(file)
char *file;

DESCRIPTION

The system is prepared to write a record in an accounting *file* for each process as it terminates. This call, with a null-terminated string naming an existing file as argument, turns on accounting; records for each terminating process are appended to *file*. An argument of 0 causes accounting to be turned off.

The accounting file format is given in *acct(5)*.

SEE ALSO

acct(5), sa(1)

DIAGNOSTICS

On error -1 is returned. The file must exist and the call may be exercised only by the super-user. It is erroneous to try to turn on accounting when it is already on.

BUGS

No accounting is produced for programs running when a crash occurs. In particular nonterminating programs are never accounted for.

ASSEMBLER (PDP-11)

(acct = 51.)
sys acct; file

NAME

alarm — schedule signal after specified time

SYNOPSIS

alarm(seconds)
unsigned seconds;

DESCRIPTION

Alarm causes signal SIGALRM, see *signal(2)*, to be sent to the invoking process in a number of seconds given by the argument. Unless caught or ignored, 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 2147483647 seconds.

The return value is the amount of time previously remaining in the alarm clock.

SEE ALSO

pause(2), signal(2), sleep(3)

ASSEMBLER (PDP-11)

(alarm = 27.)

(seconds in r0)

sys alarm

(previous amount in r0)

NAME

brk, *sbrk*, *break* — change core allocation

SYNOPSIS

char *brk(addr)

char *sbrk(incr)

DESCRIPTION

Brk 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 on the PDP11, 256 bytes on the Interdata 8/32, and 512 bytes on a VAX-11). 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.

In the alternate function *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(2), *malloc*(3), *end*(3)

DIAGNOSTICS

Zero is returned if the break could be set; -1 if the program requests more memory than the system limit or if too many segmentation registers would be required to implement the break.

BUGS

Setting the break in the range 0177701 to 0177777 (on the PDP11) is the same as setting it to zero.

ASSEMBLER (PDP-11)

(*break* = 17.)

sys break; addr

Break performs the function of *brk*. The name of the routine differs from that in C for historical reasons.

NAME

chdir — change default directory

SYNOPSIS

```
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, the starting point for path names not beginning with '/'.

SEE ALSO

cd(1)

DIAGNOSTICS

Zero is returned if the directory is changed; -1 is returned if the given name is not that of a directory or is not searchable.

ASSEMBLER

```
(chdir = 12.)
sys chdir; dirname
```

NAME

chmod — change mode of file

SYNOPSIS

```
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:

- 04000 set user ID on execution
- 02000 set group ID on execution
- 01000 save text image after execution
- 00400 read by owner
- 00200 write by owner
- 00100 execute (search on directory) by owner
- 00070 read, write, execute (search) by group
- 00007 read, write, execute (search) by others

If an executable file is set up for sharing (**-n** option of *ld(1)*) then mode 1000 prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. Thus when the next user of the file executes it, the text need not be read from the file system but can simply be swapped in, saving time. Ability to set this bit is restricted to the super-user since swap space is consumed by the images; it is only worth while for heavily used commands.

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(1)

DIAGNOSTIC

Zero is returned if the mode is changed; -1 is returned if *name* cannot be found or if current user is neither the owner of the file nor the super-user.

ASSEMBLER (PDP-11)

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

NAME

chown — change owner and group of a file

SYNOPSIS

```
chown(name, owner, group)
char *name;
```

DESCRIPTION

The file whose name is given by the null-terminated string pointed to by *name* has its *owner* and *group* changed as specified. Only the super-user may execute this call, because if users were able to give files away, they could defeat the (nonexistent) file-space accounting procedures.

SEE ALSO

chown(1), passwd(5)

DIAGNOSTICS

Zero is returned if the owner is changed; -1 is returned on illegal owner changes.

ASSEMBLER (PDP-11)

(chown = 16.)

```
sys chown; name; owner; group
```

CLOSE (2)

CLOSE (2)

NAME

close — close a file

SYNOPSIS

close(fildes)

DESCRIPTION

Given a file descriptor such as returned from an *open*, *creat*, *dup* or *pipe(2)* call, *close* closes the associated file. A close of all files is automatic on *exit*, but since there is a limit on the number of open files per process, *close* is necessary for programs which deal with many files.

Files are closed upon termination of a process, and certain high-numbered file descriptors are closed by *exec(2)*.

SEE ALSO

creat(2), open(2), pipe(2), exec(2)

DIAGNOSTICS

Zero is returned if a file is closed; -1 is returned for an unknown file descriptor.

ASSEMBLER (PDP-11)

(close = 6.)

(file descriptor in r0)

sys close

NAME

`creat` — create a new file

SYNOPSIS

```
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*, as modified by the process's mode mask (see *umask*(2)). Also see *chmod*(2) 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.

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(2), *close*(2), *chmod*(2), *umask* (2)

DIAGNOSTICS

The value `-1` 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; the file is a directory; there are already too many files open.

ASSEMBLER (PDP-11)

(`creat` = 8.)

sys creat; name; mode

(file descriptor in `r0`)

NAME

`dup`, `dup2` — duplicate an open file descriptor

SYNOPSIS

```
dup(fdes)  
int fdes;  
dup2(fdes, fdes2)  
int fdes, fdes2;
```

DESCRIPTION

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

In the second form of the call, *fdes* is a file descriptor referring to an open file, and *fdes2* is a non-negative integer less than the maximum value allowed for file descriptors (approximately 19). *Dup2* causes *fdes2* to refer to the same file as *fdes*. If *fdes2* already referred to an open file, it is closed first.

SEE ALSO

`creat(2)`, `open(2)`, `close(2)`, `pipe(2)`

DIAGNOSTICS

The value `-1` is returned if: the given file descriptor is invalid; there are already too many open files.

ASSEMBLER (PDP-11)

```
(dup = 41.)  
(file descriptor in r0)  
(new file descriptor in r1)  
sys dup  
(file descriptor in r0)
```

The *dup2* entry is implemented by adding 0100 to *fdes*.

NAME

execl, execv, execl, execve, execlp, execvp, exec, exece, environ — execute a file

SYNOPSIS

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

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

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

execve(name, argv, envp);
char *name, *argv[], *envp[];

extern char **environ;
```

DESCRIPTION

Exec in all its forms overlays the calling process with the named file, then transfers to the entry point of the core image of the file. There can be no return from a successful *exec*; the calling core image is lost.

Files remain open across *exec* unless explicit arrangement has been made; see *ioctl(2)*. Ignored signals remain ignored across these calls, but signals that are caught (see *signal(2)*) 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 *name* argument is a pointer to the name of the file to be executed. The pointers *arg[0]*, *arg[1]* ... address null-terminated strings. Conventionally *arg[0]* is the name of the file.

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; 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, envp)
int argc;
char **argv, **envp;
```

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 directly usable in another *execv* because *argv[argc]* is 0.

Envp is a pointer to an array of strings that constitute the *environment* of the process. Each string consists of a name, an "=", and a null-terminated value. The array of pointers is terminated by a null pointer. The shell *sh(1)* passes an environment entry for each global shell variable defined when the program is called. See *environ(5)* for some conventionally used names. The C run-time start-off routine places a copy of *envp* in the global cell *environ*, which is used by *execv* and *execl* to pass the environment to any subprograms executed by the current

program. The *exec* routines use lower-level routines as follows to pass an environment explicitly:

```
execve(file, argv, environ);
execl(file, arg0, arg1, . . . , argn, 0, environ);
```

Execlp and *execvp* are called with the same arguments as *execl* and *execv*, but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment.

FILES

/bin/sh shell, invoked if command file found by *execlp* or *execvp*

SEE ALSO

fork(2), environ(5)

DIAGNOSTICS

If the file cannot be found, if it is not executable, if it does not start with a valid magic number (see *a.out*(5)), if maximum memory is exceeded, or if the arguments require too much space, a return constitutes the diagnostic; the return value is -1 . Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed.

BUGS

If *execvp* is called to execute a file that turns out to be a shell command file, and if it is impossible to execute the shell, the values of *argv*[0] and *argv*[-1] will be modified before return.

ASSEMBLER (PDP-11)

(exec = 11.)

sys exec; name; argv

(exece = 59.)

sys exece; name; argv; envp

Plain *exec* is obsoleted by *exece*, but remains for historical reasons.

When the called file starts execution on the PDP11, 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, followed by a null pointer, followed by the pointers to the environment strings and then another null pointer. The strings themselves follow; a 0 word is left at the very top of memory.

```
sp → nargs
    arg0
    ...
    argn
    0
    env0
    ...
    envm
    0
arg0: <arg0\0>
    ...
env0: <env0\0>
    0
```

On the Interdata 8/32, the stack begins at a conventional place (currently 0xD0000) and grows upwards. After *exec*, the layout of data on the stack is as follows.

```
int    0
arg0:  byte  ...
    ...
argp0:  int    arg0
```

```

    ...
    int    0
envp0: int    env0
    ...
    int    0
%2→ space  40
    int    nargs
    int    argp0
    int    envp0
%3→

```

This arrangement happens to conform well to C calling conventions.

On a VAX-11, the stack begins at 0x80000000 and grows towards lower- numbered addresses. After *exec*, the layout of data on the stack is as follows.

```

ap →
fp →
sp → .long nargs
    .long arg0
    ...
    .long argn
    .long 0
    .long env0
    ...
    .long envn
    .long 0
arg0: .byte "arg0\0"
    ...
envn: .byte "envn\0"
    .long 0

```

NAME

exit — terminate process

SYNOPSIS

exit(status)

int status;

_exit(status)

int status;

DESCRIPTION

Exit is the normal means of terminating a process. *Exit* closes all the process's files and notifies the parent process if it is executing a *wait*. The low-order 8 bits of *status* are available to the parent process.

This call can never return.

The C function *exit* may cause cleanup actions before the final 'sys exit'. The function *_exit* circumvents all cleanup.

SEE ALSO

wait(2)

ASSEMBLER (PDP-11)

(exit = 1.)

(status in r0)

sys exit

NAME

`fork` — spawn new process

SYNOPSIS

`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 fact that the value returned in the old (parent) process contains the process ID of the new (child) process, while the value returned in the child is 0. Process ID's range from 1 to 30,000. This process ID is used by *wait(2)*.

Files open before the fork are shared, and have a common read-write pointer. In particular, this is the way that standard input and output files are passed and also how pipes are set up.

SEE ALSO

wait(2), *exec(2)*

DIAGNOSTICS

Returns `-1` and fails to create a process if: there is inadequate swap space, the user is not super-user and has too many processes, or the system's process table is full. Only the super-user can take the last process-table slot.

ASSEMBLER (PDP-11)

(`fork = 2.`)

sys fork

(new process return)

(old process return, new process ID in `r0`)

The return locations in the old and new process differ by one word. The C-bit is set in the old process if a new process could not be created.

NAME

getpid — get process identification

SYNOPSIS

getpid()

DESCRIPTION

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

SEE ALSO

mktemp(3)

ASSEMBLER (PDP-11)

(getpid = 20.)

sys getpid

(pid in r0)

NAME

getuid, getgid, geteuid, getegid — get user and group identity

SYNOPSIS

getuid()

geteuid()

getgid()

getegid()

DESCRIPTION

Getuid returns the real user ID of the current process, *geteuid* the effective user ID. 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.

Getgid returns the real group ID, *getegid* the effective group ID.

SEE ALSO

setuid(2)

ASSEMBLER (PDP-11)

(getuid = 24.)

sys getuid

(real user ID in r0, effective user ID in r1)

(getgid = 47.)

sys getgid

(real group ID in r0, effective group ID in r1)

NAME

indir, *syscall* — indirect system call

SYNOPSIS

syscall(*number*, *r0*, *r1*, *arg* ...) (PDP-11)

syscall(*number*, *arg*, ...) (VAX-11)

DESCRIPTION

Syscall performs the system call whose assembly language interface has the specified *number*, register arguments *r0* and *r1* (on the PDP-11, regardless of whether the entry point really uses them) and further arguments *arg*.

The *r0* value of the system call is returned.

DIAGNOSTICS

When the C-bit is set, *syscall* returns *-1* and sets the external variable *errno* (see *intro(2)*).

BUGS

There is no way to simulate system calls such as *pipe(2)*, which return values in register *r1*.

ASSEMBLER (PDP-11)

(*indir* = 0.)

sys *indir*; call

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

On the PDP-11, 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, *indir* returns error code *EINVAL*; see *intro(2)*.

NAME

ioctl, *stty*, *gtty* — control device

SYNOPSIS

```
#include <sgtty.h>
```

```
ioctl(fildes, request, argp)
```

```
struct sgttyb *argp;
```

```
stty(fildes, argp)
```

```
struct sgttyb *argp;
```

```
gtty(fildes, argp)
```

```
struct sgttyb *argp;
```

DESCRIPTION

ioctl performs a variety of functions on character special files (devices). The writeups of various devices in section 4 discuss how *ioctl* applies to them.

For certain status setting and status inquiries about terminal devices, the functions *stty* and *gtty* are equivalent to

```
ioctl(fildes, TIOCSETP, argp)
```

```
ioctl(fildes, TIOCGETP, argp)
```

respectively; see *tty*(4).

The following two calls, however, apply to any open file:

```
ioctl(fildes, FIOCLEX, NULL);
```

```
ioctl(fildes, FIONCLEX, NULL);
```

The first causes the file to be closed automatically during a successful *exec* operation; the second reverses the effect of the first.

SEE ALSO

stty(1), *tty*(4), *exec*(2)

DIAGNOSTICS

Zero is returned if the call was successful; -1 if the file descriptor does not refer to the kind of file for which it was intended.

BUGS

Strictly speaking, since *ioctl* may be extended in different ways to devices with different properties, *argp* should have an open-ended declaration like

```
union { struct sgttyb ...; ... } *argp;
```

The important thing is that the size is fixed by 'struct *sgttyb*'.

ASSEMBLER (PDP-11)

```
(ioctl = 54.)
```

```
sys ioctl; fildes; request; argp
```

```
(stty = 31.)
```

```
(file descriptor in r0)
```

```
stty; argp
```

```
(gtty = 32.)
```

```
(file descriptor in r0)
```

```
sys gtty; argp
```

NAME

kill — send signal to a process

SYNOPSIS

kill(pid, sig);

DESCRIPTION

Kill sends the signal *sig* to the process specified by the process number in *r0*. See *signal(2)* for a list of signals.

The sending and receiving processes must have the same effective user ID, otherwise this call is restricted to the super-user.

If the process number is 0, the signal is sent to all other processes in the sender's process group; see *tty(4)*.

If the process number is -1, and the user is the super-user, the signal is broadcast universally except to processes 0 and 1, the scheduler and initialization processes, see *init(8)*.

Processes may send signals to themselves.

SEE ALSO

signal(2), kill(1)

DIAGNOSTICS

Zero is returned if the process is killed; -1 is returned if the process does not have the same effective user ID and the user is not super-user, or if the process does not exist.

ASSEMBLER (PDP-11)

(kill = 37.)

(process number in *r0*)

sys kill; sig

NAME

link — link to a file

SYNOPSIS

```
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(1), unlink(2)

DIAGNOSTICS

Zero is returned when a link is made; -1 is returned 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 a file has too many links.

ASSEMBLER (PDP-11)

(link = 9.)

```
sys link; name1; name2
```

NAME

lseek, *tell* — move read/write pointer

SYNOPSIS

long *lseek*(*fildes*, *offset*, *whence*)

long *offset*;

long *tell*(*fildes*)

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 *whence* is 0, the pointer is set to *offset* bytes.

If *whence* is 1, the pointer is set to its current location plus *offset*.

If *whence* is 2, the pointer is set to the size of the file plus *offset*.

The returned value is the resulting pointer location.

The obsolete function *tell*(*fildes*) is identical to *lseek*(*fildes*, 0L, 1).

Seeking far beyond the end of a file, then writing, creates a gap or 'hole', which occupies no physical space and reads as zeros.

SEE ALSO

open(2), *creat*(2), *fseek*(3)

DIAGNOSTICS

−1 is returned for an undefined file descriptor, seek on a pipe, or seek to a position before the beginning of file.

BUGS

Lseek is a no-op on character special files.

ASSEMBLER (PDP-11)

(*lseek* = 19.)

(file descriptor in r0)

sys *lseek*; *offset1*; *offset2*; *whence*

Offset1 and *offset2* are the high and low words of *offset*; r0 and r1 contain the pointer upon return.

NAME

`mknod` — make a directory or a special file

SYNOPSIS

```
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 protection part of the mode is modified by the process's mode mask; see *umask(2)*). The first block pointer of the i-node is initialized from *addr*. For ordinary files and directories *addr* is normally 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(1)`, `mknod(1)`, `filsys(5)`

DIAGNOSTICS

Zero is returned if the file has been made; `-1` if the file already exists or if the user is not the super-user.

ASSEMBLER (PDP-11)

(`mknod` = 14.)

sys mknod; name; mode; addr

NAME

mount, umount — mount or remove file system

SYNOPSIS

mount(special, name, rwflag)

char *special, *name;

umount(special)

char *special;

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. *Name* must be a directory (unless the root of the mounted file system is not a directory). 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.

Umount announces to the system that the *special* file is no longer to contain a removable file system. The associated file reverts to its ordinary interpretation.

SEE ALSO

mount(1)

DIAGNOSTICS

Mount returns 0 if the action occurred; -1 if *special* is inaccessible or not an appropriate file; if *name* does not exist; if *special* is already mounted; if *name* is in use; or if there are already too many file systems mounted.

Umount returns 0 if the action occurred; -1 if the special file is inaccessible or does not have a mounted file system, or if there are active files in the mounted file system.

BUGS

If a file containing holes (unallocated blocks) is read, even on a file system mounted read-only, the system will attempt to fill in the holes by writing on the device.

ASSEMBLER (PDP-11)

(mount = 21.)

sys mount; special; name; rwflag

(umount = 22.)

sys umount; special

NAME

mpx — 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 in 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/sgtty.h*. The semantics of listener mode are that all file names discovered by *open(2)* to have the syntax *system!pathname* (see *uucp(1)*) 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 nonexistent 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/sgtty.h

SEE ALSO

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(1)*, 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 performance line disciplines are needed. The maximum tree depth restriction is not really checked. 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.

NAME

`nice` — set program priority

SYNOPSIS

`nice(incr)`

DESCRIPTION

The scheduling priority of the process is augmented by *incr*. Positive priorities get less service than normal. Priority 10 is recommended to users who wish to execute long-running programs without flak from the administration.

Negative increments are ignored except on behalf of the super-user. The priority is limited to the range -20 (most urgent) to 20 (least).

The priority of a process is passed to a child process by *fork(2)*. For a privileged process to return to normal priority from an unknown state, *nice* should be called successively with arguments -40 (goes to priority -20 because of truncation), 20 (to get to 0), then 0 (to maintain compatibility with previous versions of this call).

SEE ALSO

`nice(1)`

ASSEMBLER (PDP-11)

(`nice = 34.`)

(priority in `r0`)

`sys nice`

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 file is positioned at the beginning (byte 0). The returned file descriptor must be used for subsequent calls for other input-output functions on the file.

SEE ALSO

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

DIAGNOSTICS

The value -1 is returned 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.

ASSEMBLER (PDP-11)

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

PAUSE (2)

PAUSE (2)

NAME

pause — stop until signal

SYNOPSIS

pause()

DESCRIPTION

Pause never returns normally. It is used to give up control while waiting for a signal from *kill(2)* or *alarm(2)*.

SEE ALSO

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

ASSEMBLER (PDP-11)

(pause = 29.)

sys pause

NAME

pipe — create an interprocess channel

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 *fildes*[1] up to 4096 bytes of data are buffered before the writing process is suspended. A read using the descriptor *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) returns an end-of-file.

SEE ALSO

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

DIAGNOSTICS

The function value zero is returned if the pipe was created; -1 if too many files are already open. A signal is generated if a write on a pipe with only one end is attempted.

BUGS

Should more than 4096 bytes be necessary in any pipe among a loop of processes, deadlock will occur.

ASSEMBLER (PDP-11)

(pipe = 42.)

sys pipe

(read file descriptor in r0)

(write file descriptor in r1)

NAME

profil — execution time 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

monitor(3), prof(1)

ASSEMBLER (PDP-11)

(profil = 44.)

sys profil; buff; bufsiz; offset; scale

NAME

ptrace — process trace

SYNOPSIS

```
#include <signal.h>
```

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

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. 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 at location *addr* as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal that caused the stop should be ignored, or that value fetched out of the process's image indicating which signal caused the stop. If *addr* is (int *)1 then execution continues from where it stopped.
- 8 The traced process terminates.
- 9 Execution continues as in request 7; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP. (On the PDP-11 and VAX-11 the T-bit is used and just one instruction is executed; on the Interdata the stop does not take place until a store instruction is executed.) This is part of the mechanism for implementing breakpoints.

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 termi-

nation.

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 SIGTRAP.

On the Interdata 8/32, 'word' means a 32-bit word and 'even' means 0 mod 4. On a VAX-11, 'word' also means a 32-bit integer, but the 'even' restriction does not apply.

SEE ALSO

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

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

On the Interdata 8/32, 'as soon as possible' (request 7) means 'as soon as a store instruction has been executed.'

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*, *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 typewriter at most one line will be returned. In any event the number of characters read is returned.

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

SEE ALSO

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

DIAGNOSTICS

As mentioned, 0 is returned when the end of the file has been reached. If the read was otherwise unsuccessful the return value is -1. Many conditions can generate an error: physical I/O errors, bad buffer address, preposterous *nbytes*, file descriptor not that of an input file.

ASSEMBLER (PDP-11)

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

NAME

setuid, setgid — set user and group ID

SYNOPSIS

setuid(uid)

setgid(gid)

DESCRIPTION

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

SEE ALSO

getuid(2)

DIAGNOSTICS

Zero is returned if the user (group) ID is set; -1 is returned otherwise.

ASSEMBLER (PDP-11)

(setuid = 23.)

(user ID in r0)

sys setuid

(setgid = 46.)

(group ID in r0)

sys setgid

NAME

signal — catch or ignore signals

SYNOPSIS

```
#include <signal.h>

(*signal(sig, func))();
(*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 a *signal* call allows them either to be ignored or to cause an interrupt to a specified location. Here is the list of signals with names as in the include file.

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction (not reset when caught)
SIGTRAP	5*	trace trap (not reset when caught)
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	floating point exception
SIGKILL	9	kill (cannot be caught or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
	16	unassigned

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

If *func* is SIG_DFL, the default action for signal *sig* is reinstated; this default is termination, sometimes with a core image. If *func* is SIG_IGN the signal is ignored. Otherwise when the signal occurs *func* will be called with the signal number as argument. A return from the function will continue the process at the point it was interrupted. Except as indicated, a signal is reset to SIG_DFL after being caught. Thus if it is desired to catch every such signal, the catching routine must issue another *signal* call.

When a caught signal occurs during certain system calls, the call terminates prematurely. In particular this can occur during a *read* or *write*(2) on a slow device (like a terminal; but not a file); and during *pause* or *wait*(2). 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 an error status. The user's program may then, if it wishes, re-execute the call.

The value of *signal* is the previous (or initial) value of *func* for the particular signal.

After a *fork*(2) the child inherits all signals. *Exec*(2) resets all caught signals to default action.

SEE ALSO

kill(1), kill(2), ptrace(2), setjmp(3)

DIAGNOSTICS

The value (int) - 1 is returned if the given signal is out of range.

BUGS

If a repeated signal arrives before the last one can be reset, there is no chance to catch it.

The type specification of the routine and its *func* argument are problematical.

On the VAX-11, odd values for *func* are the same as SIG_IGN.

ASSEMBLER (PDP-11)

(signal = 48.)

sys signal; sig; label

(old label in r0)

If *label* is 0, default action is reinstated. 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.

NOTES (UNIX/32V)

The following defines the mapping of hardware traps to signals:

Arithmetic traps:

Integer overflow	SIGFPE
Integer division by zero	SIGFPE
Floating overflow	SIGFPE
Floating underflow	SIGFPE
Floating division by zero	SIGFPE
Decimal division by zero	SIGFPE
Decimal overflow	SIGFPE
Subscript-range	SIGFPE

Access control (i.e. protection violation)

except length violation SIGBUS

Translation not valid, and

Length access control SIGSEGV

Reserved instruction SIGILL

Customer-reserved instr. SIGEMT

Reserved operand SIGILL

Reserved addressing SIGILL

Trace pending SIGTRAP

Bpt instruction SIGTRAP

Compatibility-mode SIGEMT

Chme SIGSEGV

Chms SIGSEGV

Chmu SIGBUS

NAME

stat, fstat — get file status

SYNOPSIS

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
stat(name, buf)
```

```
char *name;
```

```
struct stat *buf;
```

```
fstat(fildes, buf)
```

```
struct stat *buf;
```

DESCRIPTION

Stat obtains detailed information about a named file. *Fstat* obtains the same information about an open file known by the file descriptor from a successful *open*, *creat*, *dup* or *pipe(2)* call.

Name points to a null-terminated string naming a file; *buf* is the address of a 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 searchable. The layout of the structure pointed to by *buf* as defined in *<stat.h>* is given below. *St_mode* is encoded according to the '#define' statements.

```
struct stat
{
    dev_t  st_dev;
    ino_t  st_ino;
    unsigned short st_mode;
    short  st_nlink;
    short  st_uid;
    short  st_gid;
    dev_t  st_rdev;
    off_t  st_size;
    time_t st_atime;
    time_t st_mtime;
    time_t st_ctime;
};
```

```
#define S_IFMT      0170000      /* type of file */
#define S_IFDIR     0040000      /* directory */
#define S_IFCHR     0020000      /* character special */
#define S_IFBLK     0060000      /* block special */
#define S_IFREG     0100000      /* regular */
#define S_IFMPC     0030000      /* multiplexed char special */
#define S_IFMPB     0070000      /* multiplexed block special */
#define S_ISUID     0004000      /* set user id on execution */
#define S_ISGID     0002000      /* set group id on execution */
#define S_ISVTX     0001000      /* save swapped text even after use */
#define S_IREAD     0000400      /* read permission, owner */
#define S_IWRITE    0000200      /* write permission, owner */
#define S_IEXEC     0000100      /* execute/search permission, owner */
```

The mode bits 0000070 and 0000007 encode group and others permissions (see *chmod(2)*). The defined types, *ino_t*, *off_t*, *time_t*, name various width integer values; *dev_t* encodes major and minor device numbers; their exact definitions are in the include file *<sys/types.h>* (see *types(5)*).

When *fildev* is associated with a pipe, *fstat* reports an ordinary file with an i-node number, restricted permissions, and a not necessarily meaningful length.

st_atime is the file was last read. For reasons of efficiency, it is not set when a directory is searched, although this would be more logical. *st_mtime* is the time the file was last written or created. It is not set by changes of owner, group, link count, or mode. *st_ctime* is set both both by writing and changing the i-node.

SEE ALSO

ls(1), filsys(5)

DIAGNOSTICS

Zero is returned if a status is available; -1 if the file cannot be found.

ASSEMBLER

(stat = 18.)

sys stat; name; buf

(fstat = 28.)

(file descriptor in r0)

sys fstat; buf

NAME

stime — set time

SYNOPSIS

stime(tp)
long *tp;

DESCRIPTION

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

SEE ALSO

date(1), time(2), ctime(3)

DIAGNOSTICS

Zero is returned if the time was set; -1 if user is not the super-user.

ASSEMBLER (PDP-11)

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

NAME

stty, gtty — set and retrieve terminal modes

SYNOPSIS

```
#include <sgtty.h>

stty(fildes, arg)
struct sgttyb *arg;

gtty(fildes, arg)
struct sgttyb *arg;
```

DESCRIPTION

Gtty stores in the structure pointed to by *arg* status information about the terminal whose file descriptor is in *r0* (in the first argument).

Stty delays until the terminal is quiescent, then sets its status to agree with the structure pointed to by *arg*.

The input and output speeds are set from the first two bytes. Both system calls use the structure defined in *<sgtty.h>*:

```
struct sgttyb {
    char    sg_ispeed;
    char    sg_ospeed;
    char    sg_erase;
    char    sg_kill;
    int     sg_flags;
};
```

The *sg_ispeed* and *sg_ospeed* fields describe the input and output speeds of the device according to the following table, which corresponds to the DH-11 interface. If DC-11, DL-11 or KL-11 interfaces are used, impossible speed changes are ignored. Symbolic values in the table are as defined in *<sgtty.h>*.

B0	0	(hang up dataphone)
B50	1	50 baud
B75	2	75 baud
B110	3	110 baud
B134	4	134.5 baud
B150	5	150 baud
B200	6	200 baud
B300	7	300 baud
B600	8	600 baud
B1200	9	1200 baud
B1800	10	1800 baud
B2400	11	2400 baud
B4800	12	4800 baud
B9600	13	9600 baud
EXTA	14	External A
EXTB	15	External B

In the current configuration, only 110, 150, 300 and 1200 baud are really supported on dial-up lines. Code conversion and line control required for IBM 2741's (134.5 baud) must be implemented by the user's program. The half-duplex line discipline required for the 202 dataset (1200 baud) is not supplied; full-duplex 212 datasets work fine.

The *sg_erase* and *sg_kill* fields of the argument structure specify the erase and kill characters respectively. (Defaults are # and @.)

The *sg_flags* field of the argument structure contains several bits which determine the system's treatment of the terminal:

ALLDELAY	0177400	Delay algorithm selection
BSDELAY	0100000	Select backspace delays:
BS0	0	
BS1	0100000	
VTDELAY	0040000	Select form-feed and vertical-tab delays:
FF0	0	
FF1	0040000	
CRDELAY	0030000	Select carriage-return delays:
CR0	0	
CR1	0010000	
CR2	0020000	
CR3	0030000	
TBDELAY	0006000	Select tab delays:
TAB0	0	
TAB1	0002000	
TAB2	0004000	
TAB3	0006000	
NLDELAY	0001400	Select new-line delays:
NL0	0	
NL1	0000400	
NL2	0001000	
NL3	0001400	
EVENP	0000200	Even parity allowed on input (e. g. for M37s)
ODDP	0000100	Odd parity allowed on input
RAW	0000040	Raw mode: wake up on all characters
CRMOD	0000020	Map CR into LF; echo LF or CR as CR-LF
ECHO	0000010	Echo (full duplex)
LCASE	0000004	Map upper case to lower on input (e. g. M33)
CBREAK	0000002	Return each character as soon as typed
HUPCL	0000001	Hang up (remove 'data terminal ready') on 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 ignored but will be used for Terminet 300's.

If a form-feed/vertical tab delay is specified, it lasts for about 2 seconds.

Carriage-return delay type 1 lasts about .08 seconds and is suitable for the Terminet 300. Delay type 2 lasts about .16 seconds and is suitable for the VT05 and the TI 700. Delay type 3 is unimplemented and is 0.

New-line delay type 1 is dependent on the current column and is tuned for Teletype model 37's. Type 2 is useful for the VT05 and is about .10 seconds. Type 3 is unimplemented and is 0.

Tab delay type 1 is dependent on the amount of movement and is tuned to the Teletype model 37. Other types are unimplemented and are 0.

Characters with the wrong parity, as determined by bits 200 and 100, 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 GE TermiNet 300's and other terminals without the newline function).

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, but since none of them are part of the standard system the specifications will not be given.

SEE ALSO

stty(1), tty(4)

DIAGNOSTICS

Zero is returned if the call was successful; -1 if the file descriptor does not refer to a terminal.

ASSEMBLER (PDP-11)

(stty = 31.)

(file descriptor in r0)

sys stty; arg

(gtty = 32.)

(file descriptor in r0) **sys gtty; arg**

NAME

sync — update super-block

SYNOPSIS

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(1), update(8)

BUGS

The writing, although scheduled, is not necessarily complete upon return from *sync*.

ASSEMBLER (PDP-11)

(sync = 36.)

sys sync

NAME

time, ftime — get date and time.

SYNOPSIS

long time(0)

long time(tloc)

long *tloc;

#include <sys/types.h>

#include <sys/timeb.h>

ftime(tp)

struct timeb *tp;

DESCRIPTION

Time returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds.

If *tloc* is nonnull, the return value is also stored in the place to which *tloc* points.

The *ftime* entry fills in a structure pointed to by its argument, as defined by <sys/timeb.h>:

```
/*
 * Structure returned by ftime system call
 */
struct timeb {
    time_t    time;
    unsigned short millitm;
    short     timezone;
    short     dstflag;
};
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval, the local timezone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

SEE ALSO

date(1), stime(2), ctime(3)

ASSEMBLER (PDP-11)

(ftime = 35.)

sys ftime; bufptr

(time = 13.; obsolete call)

sys time

(time since 1970 in r0-r1)

NAME

times — get process times

SYNOPSIS

```
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(1), time(2)

ASSEMBLER (PDP-11)

```
(times = 43.)
sys times; buffer
```

NAME

umask — set file creation mode mask

SYNOPSIS

umask (complmode)

DESCRIPTION

Umask sets a mask used whenever a file is created by *creat*(2) or *mknod*(2): the actual mode (see *chmod*(2)) of the newly-created file is the logical **and** of the given mode and the complement of the argument. Only the low-order 9 bits of the mask (the protection bits) participate. In other words, the mask shows the bits to be turned off when files are created.

The previous value of the mask is returned by the call. The value is initially 0 (no restrictions). The mask is inherited by child processes.

SEE ALSO

creat(2), *mknod*(2), *chmod*(2)

ASSEMBLER (PDP-11)

(umask = 60.)

sys umask; complmode

NAME

unlink — remove directory entry

SYNOPSIS

```
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(1), link(2)

DIAGNOSTICS

Zero is normally returned; -1 indicates that the file does not exist, that its directory cannot be written, or that the file contains pure procedure text that is currently in use. Write permission is not required on the file itself. It is also illegal to unlink a directory (except for the super-user).

ASSEMBLER (PDP-11)

```
(unlink = 10.)
sys unlink; name
```

NAME

utime — set file times

SYNOPSIS

```
#include <sys/types.h>
utime(file, timep)
char *file;
time_t timep[2];
```

DESCRIPTION

The *utime* call uses the ‘accessed’ and ‘updated’ times in that order from the *timep* vector to set the corresponding recorded times for *file*.

The caller must be the owner of the file or the super-user. The ‘inode-changed’ time of the file is set to the current time.

SEE ALSO

stat (2)

ASSEMBLER (PDP-11)

```
(utime = 30.)
sys utime; file; timep
```

NAME

wait — wait for process to terminate

SYNOPSIS

wait(status)

int *status;

wait(0)

DESCRIPTION

Wait causes its caller to delay until a signal is received or 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 the case of several children several *wait* calls are needed to learn of all the deaths.

If $(\text{int})\text{status}$ is nonzero, the high byte of the word pointed to receives the low byte of the argument of *exit* when the child terminated. The low byte receives the termination status of the process. See *signal(2)* 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(2)*. 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(2), *fork(2)*, *signal(2)*

DIAGNOSTICS

Returns -1 if there are no children not previously waited for.

ASSEMBLER (PDP-11)

(wait = 7.)

sys wait

(process ID in r0)

(status in r1)

The high byte of the status is the low byte of r0 in the child at termination.

NAME

write — write on a file

SYNOPSIS

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

DESCRIPTION

A file descriptor is a word returned from a successful *open*, *creat*, *dup*, or *pipe(2)* call.

Buffer is the address of *nbytes* contiguous bytes which are written on the output file. The number of characters actually written is returned. 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(2), *open(2)*, *pipe(2)*

DIAGNOSTICS

Returns -1 on error: bad descriptor, buffer address, or count; physical I/O errors.

ASSEMBLER (PDP-11)

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