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 succesful calls, so it should be tested only after an error has occurred. There is a table of messages associated with each error, and a routine for printing the message. See *perror (III).*

The possible error numbers are not recited with each writeup in section 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 – (unused)

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 512 bytes (counting the null at the end of each argument) 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 one of the magic numbers 407 or 410.

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 – (unused)

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 was made to dismount a device on which there is an open file or some process's current directory.

17 EEXIST File exists
In existing file was mentioned in a context in which it should not have, 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 10 files can be open per process; this error occurs when the eleventh is opened.

25 ENOTTY Not a typewriter
The file mentioned in *stty* or *gtty* is not a typewriter 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!).

27 EFBIG File too large
An attempt to make a file larger than the maximum of 2048 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.

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 succesful calls, so it should be tested only after an error has occurred. There is a table of messages associated with each error, and a routine for printing the message. See *perror (III).*

The possible error numbers are not recited with each writeup in section 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 – (unused)

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 512 bytes (counting the null at the end of each argument) 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 one of the magic numbers 407 or 410.

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 – (unused)

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 was made to dismount a device on which there is an open file or some process's current directory.

17 EEXIST File exists
In existing file was mentioned in a context in which it should not have, 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 10 files can be open per process; this error occurs when the eleventh is opened.

25 ENOTTY Not a typewriter
The file mentioned in *stty* or *gtty* is not a typewriter 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!).

27 EFBIG File too large
An attempt to make a file larger than the maximum of 2048 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.

NAME

break – set program break

SYNOPSIS

(break = 17.)
sys break; addr

char *sbrk(incr)

DESCRIPTION

Break sets the system's idea of the lowest location not used by the program 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, the calling sequence is different; *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)

DIAGNOSTICS

The c-bit is set if the program requests more memory than the system limit (currently 20K words), or if more than 8 segmentation registers would be required to implement the break. From C, –1 is returned for these errors.

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
0400 read by owner
0200 write by owner
0100 execute by owner
0070 read, write, execute by group
0007 read, write, execute by others

Only the owner of a file (or the super-user) may change the 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

SYNOPSIS

(chmod = 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 changed to *owner* (a numerical user ID). Only the present owner of a file (or the super-user) may donate the file to another user. Changing the owner of a file removes the set-user-ID protection bit unless it is done by the super user or the real user ID is the new owner.

SEE ALSO

chown(I), uids(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 10 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 10 files open.

From C, a –1 return indicates an error.

-

NAME

csw – read console switches

SYNOPSIS

(csw = 38.; not in assembler)

sys csw

getcsw( )

DESCRIPTION

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

NAME

dup – duplicate an open file descriptor

SYNOPSIS

(dup = 41.; not in assembler)
(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 between 0 and 9, 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 10 open files. From C, a –1 returned value indicates an error.

NAME

exec − execute a file

SYNOPSIS

(exec = 11.
sys exec; name; args
...
name: <**...\0**>
...
args: arg1; arg2; ...; 0
arg1: <**...\0**>
arg2: <**...\0**>
...

execl(name, arg1, arg2, ..., argn, 0)
char *name, *arg1, *arg2, ..., *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
      arg1
      ...
      argn

arg1: <arg1\0>
      ...
argn: <argn\0>
```

From C, two intefaces 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)

DIAGNOSTICS

If the file cannot be found, if it is not executable, if it does not have a valid header (407 or 410 octal as first word), if maximum memory is exceeded, or if the arguments require more than 512 bytes a return from *exec* constitutes the diagnostic; the error bit (c-bit) is set. From C the returned value is –1.

BUGS

Only 512 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 the process' files 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)

DIAGNOSTICS

None.

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.*

From C, the returned value is 0 in the child process, non-zero in the parent process; however, a return of −1 indicates inability to create a new process.

SEE ALSO

wait(II), exec(II)

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 (not just negative) 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 identification

SYNOPSIS

(getgid = 47.; not in assembler)
sys getgid

getgid( )

DESCRIPTION

Getgid returns the real 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)

DIAGNOSTICS

–

-

NAME

getuid – get user identification

SYNOPSIS

(getuid = 24.)
sys getuid

getuid( )

DESCRIPTION

Getuid returns the real 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)

DIAGNOSTICS

–

NAME

gtty – get typewriter 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 typewriter 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 typewriter. From C, a –1 value is returned for an error, 0, for a successful call.

NAME

indir – indirect system call

SYNOPSIS

(indir = 0.; not in assembler)
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.

SEE ALSO

–

DIAGNOSTICS

–

NAME

kill – send signal to a process

SYNOPSIS

(kill = 37.; not in assembler)
(process number in r0)
sys kill; 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 controlling typewriter, otherwise this call is restricted to the super-user.

SEE ALSO

signal(II), kill(I)

DIAGNOSTICS

The error bit (c-bit) is set if the process does not have the same controlling typewriter and the user is not super-user, or if the process does not exist.

BUGS

Equality between the controlling typewriters of the sending and receiving process is neither a necessary nor sufficient condition for allowing the sending of a signal. The correct condition is equality of user IDs.

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

link(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. From C, a –1 return indicates an error, a 0 return indicates success.

NAME

mknod – make a directory or a special file

SYNOPSIS

(mknod = 14.; not in assembler)
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(I), 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

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.

SEE ALSO

mount(I), 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; 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 −220. The value of 16 is recommended to users who wish to execute long-running programs without flak from the administration.

The effect of this call is passed to a child process by the *fork* system call. The effect can be cancelled by another call to *nice* with a *priority* of 0.

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

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

NAME

pipe – create a pipe

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 are ignored.

SEE ALSO

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

DIAGNOSTICS

The error bit (c-bit) is set if more than 8 files are already open. From C, a –1 returned value indicates an error.

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

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

SEE ALSO

open(II), creat(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)

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.; not in assembler)
(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

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; value

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

DESCRIPTION

When the signal defined by *sig* is sent to the current process, it is to be treated according to *value*. The following is the list of signals:

1	hangup
2	interrupt
3*	quit
4*	illegal instruction
5*	trace trap
6*	IOT instruction
7*	EMT instruction
8*	floating point exception
9	kill (cannot be caught or ignored)
10*	bus error
11*	segmentation violation
12*	bad argument to sys call

If *value* is 0, the default system action applies to the signal. This is processes termination with or without a core dump. If *value* is odd, the signal is ignored. Any other even *value* specifies an address in the process where an interrupt is simulated. An RTI instruction will return from the interrupt. As a signal is caught, it is reset to 0. Thus if it is desired to catch every such signal, the catching routine must issue another *signal* call.

The starred signals in the list above cause core images if not caught and not ignored. In C, if *func* is 0 or 1, the action is as described above. If *func* is 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 simulate the RTI.

After a *fork*, the child inherits all signals. The *exec* call resets all caught signals to default action.

SEE ALSO

kill (I, II)

DIAGNOSTICS

The error bit (c-bit) is set if the given signal is out of range. In C, a –1 indicates an error; 0 indicates success.

NAME

sleep – stop execution for interval

SYNOPSIS

(sleep = 35.; not in assembler)
(seconds in r0)
sys sleep

sleep(seconds)

DESCRIPTION

The current process is suspended from execution for the number of seconds specified by the argument.

SEE ALSO

sleep (I)

DIAGNOSTICS

–

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 {
  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
 000400     read (owner)
 000200     write (owner)
 000100     execute (owner)
 000070     read, write, execute (group)
 000007     read, write, execute (others)
```

SEE ALSO

stat(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 typewriter

SYNOPSIS

(stty = 31.)
(file descriptor in r0)
sys stty; arg
...
arg: speed; 0; mode

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

DESCRIPTION

Stty sets mode bits and character speeds for the typewriter whose file descriptor is passed in r0 (resp. is the first argument to the call). First, the system delays until the typewriter is quiescent. Then the speed and general handling of the input side of the typewriter is set from the low byte of the first word of the *arg*, and the speed of the output side is set from the high byte of the first word of the *arg*. The speeds are selected from the following table. This table corresponds to the speeds supported by the DH-11 interface. The starred entries are those speeds actually supported by the DC-11 interfaces actually present; if a non-starred speed is selected, it will be ignored and the present speed left unchanged.

0	(turn off device)
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 150 and 300 baud are really supported, in that the code conversion and line control required for 2741's (134.5 baud) must be implemented by the user's program, and the half-duplex line discipline required for the 202 dataset (1200 baud) is not supplied.

The second word of the *arg* is currently unused and is available for expansion.

The third word of the *arg* sets the *mode.* It contains several bits which determine the system's treatment of the typewriter:

10000	no delays after tabs (e.g. TN 300)
200	even parity allowed on input (e. g. for M37s)
100	odd parity allowed on input
040	raw mode: wake up on all characters
020	map CR into LF; echo LF or CR as CR-LF
010	echo (full duplex)
004	map upper case to lower on input (e. g. M33)
002	echo and print tabs as spaces
001	inhibit all function delays (e. g. CRTs)

Characters with the wrong parity, as determined by bits 200 and 100, are ignored.

In raw mode, every character is passed back immediately to the program. No erase or kill processing is done; the end-of-file character (EOT), the interrupt character (DELETE) 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).

SEE ALSO

stty(I), gtty(II)

DIAGNOSTICS

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

NAME

sync – update super-block

SYNOPSIS

(sync = 36.; not in assembler)
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 *check*, *df*, etc. It is mandatory before a boot.

SEE ALSO

sync (VIII), update (VIII)

DIAGNOSTICS

–

-

NAME

time – get date and time

SYNOPSIS

(time = 13.)
sys time

time(tvec)
int tvec[2];

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)

DIAGNOSTICS

none

NAME

times – get process times

SYNOPSIS

(times = 43.; not in assembler)
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 {
        int     proc_user_time;
        int     proc_system_time;
        int     child_user_time[2];
        int     child_system_time[2];
};
```

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

SEE ALSO

time(I)

DIAGNOSTICS

–

BUGS

The process times should be 32 bits as well.

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*).

SEE ALSO

umount(I), 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

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

wait – wait for process to die

SYNOPSIS

(wait = 7.)
sys wait

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). 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. If the 040 bit of the termination status is set, a core image of the process was produced by the system.

SEE ALSO

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

DIAGNOSTICS

The error bit (c-bit) on if 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* 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 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.