NAME

intro − introduction to system calls

DESCRIPTION

Section 2 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 **−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 *ermo* is set to the error number. *Ermo* 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 inside the system (for the benefit of system-readers), and the messages available using *perror.* A short explanation is also provided.

Users needing to examine these error codes directly should include the file **/usr/include/errno.h** rather than wiring these numbers into their program.

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 th 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. Also returned for an attempt to send a message to a process that has not enabled message reception.

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 is beyond the limits of the allowed number of subdevices. It may also occur when, for example, a tape drive is not on-line, or no disk pack is loaded on a drive.

7 **E2BIG** Arg list too long
: An argument list longer than 5,120 bytes (counting the null at the end of each argument) is presented to a member of the *exec* family.

8 **ENOEXEC** Exec format error
: a request is made to ececute 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* was requested but 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 memory than the system is able to supply. This is not a temporary condition; the maximum memory size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments is such as to require mor than the existing 8 segmentation registers, or if there is not enough swap space during a *fork*.

13 **EACCES** Permission denied
: An attempt was made to access a file or some other resource in a way forbidden by the protection system.

14 **EFAULT** Memory fault
: User has supplied a non-existent address.

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

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

17 **EEXIST** File exists
: An existing file was memtioned 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 *cd*.

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, memtioning an unknown signal in *signal*, giving an unknown request to *ioctl*, passing an invalid argument list to *exec*, providing an unknown *function* argument to *sema* or *msg*, reading or writing a file for which *lseek* has returned a negative pointer, multiple system profiling was requested, or invalid math function arguments (see 3M).

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

24 **EMFILE** Too many open files
: Only 20 files can be open per process.

25 **ENOTTY** Not a typewriter
: The file mentioned in *ioctl* 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). Also an attempt to open for writing a pure- procedure program that is being executed.

27 **EFBIG** File too large
: An attempt to make a file larger than the maximum of **ULIMIT**, or 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
: 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 127 links to a file.

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

33 **ETABLE** No entries left
: One of the system tables necessary to complete the request is temporarily full, or the argement of a function in the math package (3M) is out of the domain of the function.

33 **EDOM** Math argument
: The argument of a function in the math package (3M) is out of the domain of the function. This error number is used by certain programs that were transported from UNIX/TS to CB-UNIX after that error number was already in use for another purpose in CB-UNIX (see 33 **ETABLE** above).

34 **EFUNC** Invalid function
: An attempt to perform an invalid operation, or the value of a function in the math package (3M) is not representable within machine precision.

34 **ERANGE** Result too large
: The value of a function in the math package (3M) is not representable within machine precision. This error number is also used for 34 **EFUNC** above. See the note under 33 **EDOM** above.

35 **ENOMSG** No message
: A message of the requested type is not on the message queue.

36 **ENOALOC** Resource not allocated
: An attempt was made either to use a facility that must first be allocated (e.g., *recvw* without first enabling messages) or th allocate a facility in a way other than for its intended use.

37 **ELOCK** Locking error

An attempt was made to unlock a process or text that was not locked, or an attempt was made to lock the process/text when the text/process was already locked (the locking specifications are mutually exclusive). Other possibilities are that a locked process tried to *exec*, or that *sprofil* could not lock the user buffer in memory.

SEE ALSO

intro (3)

ASSEMBLER

as /usr/include/sys.s file

The PDP-11 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 expect these registers to be preserved. An erroneous call is always indicated by turning on the c-bit of the condition codes. The error number is then 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.

For the *syscb* and *utssys* groups of system calls, the call's number is passed in r1 and the first argument to the call, if there is one, is passed in r0.

OLD C COMPILER

The manual pages in sections 2 and 3 with names ending in ":o" are system calls and functions that are for use with the old C compiler (*occ*) and/or release 1 of CB-UNIX. When using the old C compiler, old system calls are used automatically via special "stamping" of the version number in the load files in the compiled programs (see *stamp*(1)). For system calls that do not exist in the new compiler, the old system routines are called. For system calls that have a *sysent* index identical to a release 2 system call, the stamping determines which will be executed. The ":o" routines are kept around primarily for the commands in section 1 of this manual, which are still compiled with the old C compiler, and for user routines from version 1 systems that were compiled with the old C compiler (which was the new C compiler under the old system). The load files for release 1 user programs have the inherent stamping that will cause the proper system calls and library routines to be executed.

Users should eventually change and recompile programs that require any ":o" routines, so that all programs use the new C compiler and current system calls - the current *occ* will not be available in release 3 of CB-UNIX. Of course, any new programs should be written to use the new C compiler, if at all possible. It is anticipated that the CB-UNIX Systems Group will follow its own advice and change those commands in section 1 that still require the old compiler.

NAME
access − determine accessibility of file

SYNOPSIS
```
int access (name, mode)
char *name;
int mode;
```

DESCRIPTION
Access checks the given file *name* for accessibility according to *mode*, which is 4 (read), 2 (write) or 1 (execute) or a combination thereof.

An appropriate error indication is returned if *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
(access = 33.)
sys access; name; mode

NAME

acct − turn accounting on/off

SYNOPSIS

acct (name)
char *name;

DESCRIPTION

System per process accounting is started and stopped by the *acct* system call. When system accounting is active, a record in shell accounting format (see *sa*(1)) is placed at the end of the file *name* for each process that terminates. *Name* is a pointer to a null terminated string of ASCII characters that represents the file's pathname. System accounting is terminated by invoking *acct* with a zero value for *name*.

Only the super-user may start and stop system accounting.

SEE ALSO

accton(1), sa(1), acct(5), intro(2)

DIAGNOSTICS

The error bit (c-bit) is set if the user is not super-user or if an attempt is made to start accounting when it is already active. System accounting must have been enabled at system generation time, else error ENODEV will be returned. From C, a −1 return indicates an error.

ASSEMBLER

(acct = 51.; not in assembler)
sys acct; name

NAME

alarm − schedule signal after specified time

SYNOPSIS

unsigned 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 canceled. Because the clock has a 1-second resolution, the signal may occur up to one second early; because of scheduling delays, resumption of execution of when the signal is caught may be delayed an arbitrary amount. The longest specifiable delay time is 65535 seconds.

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

SEE ALSO

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

ASSEMBLER

(alarm = 27.)
(seconds in r0)
sys **alarm**
(previous amount in r0)

NAME

break, brk, sbrk – change memory allocation

SYNOPSIS

char *sbrk (incr)
char *brk (addr)

DESCRIPTION

Brk 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.

The call to *sbrk* should normally be used; *incr* more bytes are added to the program's data space and a pointer to the start of the new area is returned.

The assembler call *break* call will act exactly like the *brk* call described above. A zero is return upon success.

When a program begins execution via *exec* the break is set at the highest location defined by the program's data area (for programs running without separated I&D space the highest location is the sum of the text and data space sizes). The amount of space available for *break* to grab is thus the difference between the highest program location and the bottom of the stack. By definition, the amount of space reserved for the stack (but not necessarily allocated) is 4096 words.

Also, be careful when using *maus* together with *break*. *Maus* has the effect of reducing the amount of space available for allocation with *break*.

SEE ALSO

maus(2), exec(2), malloc(3)

DIAGNOSTICS

The c-bit is set whenever it is impossible to grant the memory request. From C, –1 is returned for these errors.

BUGS

It is possible to reference memory past the end of the *break* without incurring an error. In fact, it is possible to overflow normal references (for instance an array subscript gone wild) into the stack without a memory fault occuring.

ASSEMBLER

(break = 17.)
sys break; addr

NAME

chdir − change working 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.

SEE ALSO

chdir(1), chroot(2)

DIAGNOSTICS

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

ASSEMBLER

(chdir = 12.)
sys chdir; dirname

NAME
chgrp − change group

SYNOPSIS
chgrp (name, group)
char *name;

DESCRIPTION
The file whose name is given by the null-terminated string pointed to by *name* has its group changed to *group* (a numerical group ID). Only the present owner of a file (or the super-user) may change the group of a file. Changing the group of a file removes the set-group-ID protection bit unless it is done by the super user.

Chgrp has been dropped from the new version of the library. Use *chown*(2) instead.

SEE ALSO
chgrp(1), chown(2)

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

NAME

chmod − change mode of file

SYNOPSIS

int chmod (name, mode)
char *name;
int mode;

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** or **−i** 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 01000 mode.

SEE ALSO

chmod(1), umask(2)

DIAGNOSTIC

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

ASSEMBLER

(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 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 and set-group-ID protection bits, unless it is done by the super user.

SEE ALSO

chown(1), chgrp(2), passwd(5)

DIAGNOSTICS

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

ASSEMBLER

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

NAME

chown − change owner

SYNOPSIS

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.

SEE ALSO

chown(1), chgrp(2), passwd(5)

DIAGNOSTICS

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

NAME

chroot − change root directory

SYNOPSIS

chroot (rootname)
char *rootname;

DESCRIPTION

Rootname is the address of the pathname of a directory, terminated by a null byte. *Chroot* causes this directory to become the process' root directory. This means that any references to file names beginning with slash are not relative to the real root of the UNIX file system, but relative to the new root directory specified in this system call. The current working directory remains unchanged. Notice, however, that a chdir to slash ("/") following the *chroot* system call will change the working directory to the new root directory. Arguments to *chroot* are always absolute: no special meaning is given to initial slashes even if a *chroot* is currently in effect.

This system call is restricted to the super-user.

SEE ALSO

chroot(1)

DIAGNOSTICS

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

ASSEMBLER

(chroot = 61.)
sys chroot; dirname

NAME

close − close a file

SYNOPSIS

int close (fildes)
int fildes;

DESCRIPTION

Given *fildes,* a file descriptor such as returned from an *open*, *creat*, *dup* or *pipe* call, *close* closes the associated file. A close of all files is automatic on *exit*, but since there is a limit of 20 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 may also be set to be closed automatically on *exec*(2) − see *ioctl*(2).

SEE ALSO

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

DIAGNOSTICS

Zero is returned if the file is closed successfully; −1 is returned for an unknown file descriptor.

ASSEMBLER

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

NAME

creat − create a new file

SYNOPSIS

int creat (name, mode)
char *name;
int mode;

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; the file will be opened for writing even if the *mode* does 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

chmod(2), close(2), umask(2), write(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 20 files open.

ASSEMBLER

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

NAME

dup − duplicate an open file descriptor

SYNOPSIS

dup (fildes)
struct { char lobyte; char hibyte; } 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 original file descriptor must be placed in the low byte of *fildes*, i.e. *fildes.lobyte*, the high byte of i.e. *fildes.hibyte*, must be 0. The new file descriptor is returned. Normally the first available file descriptor is returned, however, the system can be forced to assign file descriptors starting at some number other than zero by setting the high byte of *fildes*, i.e. *fildes.hibyte*, to the desired starting point. The system attempts to find a non-allocated file descriptor ⩾ *fildes.hibyte*. If all of the file descriptors beyond the start point are used, the user is returned an error even if there are file descriptors with smaller values available. A process may have up to 20 file descriptors open at a time and the file descriptors will be assigned as numbers from zero to nineteen.

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

SEE ALSO

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

DIAGNOSTICS

The error bit (c-bit) is set if: the given file descriptor is invalid; there are already too many open files; there are no more file descriptors beyond the value specified in the high byte of *fildes*. From C, a −1 returned value indicates an error.

ASSEMBLER

(dup = 41.; not in assembler)
(file descriptor in r0)
sys **dup**

NAME

execl, execv, execle, execve, execlp, execvp − execute a file

SYNOPSIS

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

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

int execle (name, arg0, arg1, ..., argn, 0, envp)
char *name, *arg0, *arg1, ..., *argn, *envp[ ];

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

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

int execvp (name, argv)
char *name, *argv[ ];
```

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.

File descriptors ordinarily remain open across *exec*, but may be requested to be automatically closed (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 or 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 and IDs are 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*(7) for some conventionally used names. The C run-time start-off routine places a copy of *envp* in a global cell:

extern char **environ;

that 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);
execle (file, arg0, arg1, . . . , argn, 0, environ);

Execvp and *execlp* are called with the same arguments as *execv* and *execl,* 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, or the value specified by the shell variable **$SHELL**, invoked if command file found by *execlp* or *execvp*

SEE ALSO

ioctl(2), fork(2), getenv(3C), environ(7)

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, if it is a pure-procedure program which is currently open for reading or writing, 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.

ASSEMBLER

(exec = 11.)
sys exec; name; argv

(exece = 59.)
sys exece; name; argv; envp

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

When the called file starts execution, 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
```

This arrangement happens to conform well to C calling conventions.

NAME

exec, execl, execv, exect − execute a file

SYNOPSIS

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

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

exect (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 except that all "auto-close" files are closed (see *dup*(2) and *open*(2)). Ignored signals remain ignored across *exec*, but signals that are caught are reset to their default values. All *maus* descriptors remain open but no *maus* segments remain attached (see *maus*(2)).

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

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

The first argument to *exec* is a pointer to the name of the file to be executed. The second is the address of a null-terminated list of pointers to arguments to be passed to the file. Conventionally, the first argument is the name of the file. Each pointer addresses a string terminated by a null byte.

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

```
sp−> nargs
     arg0
     ...
     argn
     0

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

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

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

Exect, if successful, causes the trace bit (020) to be turned on in the program status word. It is otherwise identical to *execv*.

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 directly usable in another *execv*, since *argv*[*argv*] is 0. There is a new version of *exec*.

SEE ALSO

call(2), fork(2), open(2), dup(2), maus(2), signal(2)

DIAGNOSTICS

If the file cannot be found, if it is not executable, if it does not have a valid header (407, 410, or 411 octal as first word), if maximum memory is exceeded, or if the arguments require more than 512 bytes a return from *exec* constitutes the diagnostic; the error bit (c-bit) is set. Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed. From C the returned value is −1.

NAME

exit − terminate process

SYNOPSIS

exit (status)
struct { char lobyte; char hibyte; } status;

_exit(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, *status.lobyte*, is available as status to the parent process via *wait*.

There are two C callable versions. *Exit* calls the user definable routine **_cleanup** to perform any user defined cleanup actions; then it does an *exit*. The C library version of *_cleanup* - which is used if the user does not supply his own - simply returns. The other version, *_exit*, exits without calling anything. It is provided so that users may write their own version of *exit*.

When a process dies, e.g. via *exit*, the child death signal, **SIGCLD**, is sent to its parent (see *signal*(2)).

This call can never return.

SEE ALSO

wait(2), signal(2), fclose(3S)

ASSEMBLER

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

NAME

fcntl − file control

SYNOPSIS

int fcntl (fildes, request, argument)
int fildes, request, argument;

DESCRIPTION

This function provides for control over open files. The *requests* available are:

0 − duplicate *fildes* as the lowest-numbered available file descriptor not less than *argument*. Returns the new file descriptor, or **−1** if no appropriate file descriptors are available, or if *fildes* is not open. The flags (see below) are set to zero.

1 − Get *fildes* flags where a **0** return means a normal file and **1** means that the file should be closed on *exec*.

2 − Set *fildes* flags to *argument* (**0** or **1** as above). Returns *fildes* if successful, otherwise **−1**.

EXAMPLES

The following pairs of calls are equivalent:

dup(x) = fcntl(x, 0, 0)
rename(x, y) = close(y), fcntl(x, 0, y), close(x)

NOTE

The future of this command is unsure. *Ioctl*(2) should be used for auto-close purposes.

SEE ALSO

exec(2), open(2), close(2), ioctl(2)

ASSEMBLER

(fcntl = 62.)
(fildes in r0)
sys fcntl; request; argument
(result in r0)

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 return location and the fact that r0 in the old (parent) process contains the process ID of the new (child) process. (In the new process, r0 contains a 0). This process ID is used by *wait*(2).

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

In the child process, the external integer **par_uid** contains the process ID of the parent process.

From C, the returned value is 0 in the child process. The value returned to the parent process is the child's process ID; however, a return of −1 indicates inability to create a new process.

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

SEE ALSO

wait(2), exec(2)

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.

ASSEMBLER

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

The return locations in the old and new processes differ by one word.

NAME
fstat — get status of open file

SYNOPSIS
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. The *inode* structure is defined in *stat*(2).

SEE ALSO
stat(2)

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.

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

NAME

ftime − get date and time

SYNOPSIS

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

ftime (tp)
struct timeb *tp;

DESCRIPTION

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

```
/*              @(#)/usr/src/ucb/sys/timeb.h    3.1             */

/*
 * 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, the number of milliseconds (less than 1000) since the last second, the local timezone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Savings Time applies locally during the appropriate part of the year.

SEE ALSO

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

ASSEMBLER

(ftime = 35.)
sys ftime; bufptr

Add

NAME
getcsw — read console switches

SYNOPSIS
getcsw ( )

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

ASSEMBLER
(syscb = 45.; getcsw = 1)
sys syscb; getcsw

NAME
getgid − get group identification

SYNOPSIS
getgid ( )

DESCRIPTION
Getgid returns a word defined as follows:

struct { char lobyte; char hibyte; } val;

Val.lobyte returns the real group ID and *val.hibyte* returns the effective group ID of the current process. The real group ID identifies the group of the person who is logged in, in contradistinction to the effective group ID, which determines his access permission at the moment. It is thus useful to programs which operate using the "set group ID" mode, to find out who invoked them.

Getgid is obsolete − use *getuid*(2) instead in new code.

SEE ALSO
setgid(2), getuid(2)

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

NAME

getpid, getppid — get process identification

SYNOPSIS

getpid ( )
getppid ( )

DESCRIPTION

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

Getppid returns the process ID of the parent of the current process. Note that a return of **1** from this call indicates that the parent process has terminated and the current process has been inherited by the initialization process.

SEE ALSO

fork(2), init(1M)

ASSEMBLER

(getpid = 20.; not in assembler)
sys getpid
(pid in r0)
(parent pid in r1)

NAME

getu − get selected user block information

SYNOPSIS

getu (offset, buffer, nbytes)
char *buffer;

DESCRIPTION

Getu is used to copy information from the current process' user control block into the user's address space. *Nbytes* are copied into user's space starting at the address specified by *buffer*. The copy begins *offset* bytes into the control block. All three arguments must be even numbers. The number of bytes actually transferred is returned as the result.

DIAGNOSTICS

The error bit (c-bit) is set for an illegal request. If the actual number of bytes transferred is not equal to the requested number, an error has occurred. From C, a −1 return indicates an error.

ASSEMBLER

add

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 in r0 and the effective user ID in r1. 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.

In C, *getuid* returns the real user ID, *geteuid* the effective user ID.

Getgid returns the real group ID in r0, the effective group ID in r1.

In C, *getgid* returns the real group ID, *getegid* the effective group ID.

SEE ALSO

setuid(2)

ASSEMBLER

(getuid = 24.)
sys getuid

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

NAME

getuid − get user identification

SYNOPSIS

getuid ( )

DESCRIPTION

Getuid returns a word with the following layout:

struct { char lobyte; char hibyte; } val;

Val.lobyte contains the real user ID and *val.hibyte* contains the effective user ID of the current process. The real user ID identifies the person who is logged in, in contradistinction to the effective user ID, which determines his access permission at the moment. It is thus useful to programs which operate using the "set user ID" mode, to find out who invoked them.

SEE ALSO

setuid(2)

NAME

indir − indirect system call

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.

ASSEMBLER

(indir = 0.; not in assembler)
sys indir; syscall

NAME

ioctl − control device

SYNOPSIS

#include <sys/ioctl.h>

int ioctl (fildes, request, argp)
int fildes, request;
struct *argp;

DESCRIPTION

Ioctl manipulates the file or device indicated by *fildes* as specified by *request*. The requests and the kinds of things they can access are:

TIOCGETD, TIOCSETD
Get/set line discipline. *Argp* points to a structure containing an integer with a valid line discipline indicator integer.

TIOCHPCL Hang up on last close. *Argp* indicates whether this feature should be turned on or off.

TIOCSETO, TIOCGETO
Get/set "other" bits. *Argp* contains a word with bits indicating which "other" bits are to be set/reset or interrogated. This request is essentially an extension of the old *stty/gtty* system call that allows transmission/response to xon/xoff, half duplex line, no-hangup, excluding future device opens, no sleeping if not ready, and non-standard tty escapes and kills.

TIOCGETP, TIOCSETP
These are equivalent to **gtty(fildes, argp)** and **stty(fildes, argp)**. They allow terminal (tty) characteristics to be set and examined. These include terminal input and output speed, the erase character and kill character, and mode flags. The allowed mode flags include hangup on last close, map tabs to spaces, upper case only, character echo, cr/lf mode, raw character input, parity, and delay on tabs, new lines, backspace, carriage return, and vt delay. Note that setting input speed to zero on a dh or dz line will disable the line by dropping the Data Terminal Ready(DTR) bit for the line.

TIOCSETN Equivalent to old *stty* with *noflush*.

TIOCEXCL, TIOCNXCL
Get/clear the exclude bit, which disallows future opens on the device.

TIOCTSTP Stop toggle transmit.

DIOCGETT, DIOCSETT
Get/set terminal parameters. These include terminal type, current cursor row and column (get only), variable row, last row, and terminal flags. The flags include special newline, auto newline on column 80, last column of last row special, echo of terminal cursor control, and not sending escape sequences to the user. It is used primarily for CRT terminals.

DIOCSETS Set spy mode. All output directed to the terminal specified by *fildes* will be copied to the terminal of the process performing the *ioctl*. Only one spy operation may be active in the entire system at any time. The spy continues until explicitly turned off. Currently, spy is only effective on lines using the STD_LTYPE line discipline and is restricted to the super-user.

FIOCLEX, FIONCLEX
Set/clear auto close for a file. If auto close is set, then the file will not be passed to children across an *exec*.

FIOSPIPE, FIOGPIPE
Get/set pipe sleep flags. This enables/disables sleeping on reads/writes to a pipe, to avoid roadblocking. Normally, reads are blocking and writes are not.

VIOCGETD, VIOCSETD
Get/set versatec parameters.

There are also requests for the multiplexor (see *mpx*(2), *mpxio*(5) and **<sys/mx.h>**). In general, each line discipline has a unique header file which defines the line discipline number and format of the structure to be used with **DIOCGETP** and **DIOCSETP** requests.

The proper names for all these flags and other requests not currently used are contained in **<sys/ioctl.h>**, which is included here:

```
/*              @(#)ioctl.h     3.5             */
/*
 * structure of arg for ioctl TIOCSETP and TIOCGETP
 */
struct          ttiocb {
                char            ioc_ispeed;
                char            ioc_ospeed;
                char            ioc_erase;
                char            ioc_kill;
                short           ioc_flags;
};

/*
 * structure for old stty and gtty system calls.
 */
struct          sgttyb          {
                char            sg_ispeed;      /* input speed */
                char            sg_ospeed;      /* output speed */
                char            sg_erase;       /* erase character */
                char            sg_kill;        /* kill character */
                short           sg_flags;       /* mode flags */
};

/*
 * tty ioctl commands
 */
#define         TIOCGETD        (('t'<<8)|0)    /* get line discipline */
#define         TIOCSETD        (('t'<<8)|1)    /* set line discipline */
#define         TIOCHPCL        (('t'<<8)|2)    /* hangup on last close */
#define         TIOCMODG        (('t'<<8)|3)
#define         TIOCMODS        (('t'<<8)|4)
#define         TIOCSETO        (('t'<<8)|6)    /* set other bits */
#define         TIOCGETO        (('t'<<8)|7)    /* get other bits */
#define         TIOCGETP        (('t'<<8)|8)    /* gtty */
#define         TIOCSETP        (('t'<<8)|9)    /* stty */
#define         TIOCSETN        (('t'<<8)|10)   /* stty - no flush */
#define         TIOCEXCL        (('t'<<8)|13)   /* set exclude */
#define         TIOCNXCL        (('t'<<8)|14)   /* clr exclude */
#define         TIOCHMOD        (('t'<<8)|15)
#define         TIOCTSTP        (('t'<<8)|16)   /* toggle transmit stop */
#define         DIOCGETP        (('d'<<8)|8)    /* get discipline parameters */
#define         DIOCSETP        (('d'<<8)|9)    /* set discipline parameters */
#define         DIOCSETT        (('d'<<8)|10)   /* set terminal info */
```