NAME

intro – introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value. This is almost always –1; the individual descriptions 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 been indicated.

Each system call description attempts to list all possible error numbers. The following is a complete list of the error numbers and their names as defined in **<errno.h>**.

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

No process can be found corresponding to that specified by *pid* in *kill* or *ptrace*.

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 has occurred. 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 on-line or no disk pack is loaded on a drive. On local terminals, it may indicate that the host terminal lacks the specified channel; for example, opening tp2033, when tty033 refers to a TM31 Terminal.

7 E2BIG Arg list too long

An argument list longer than 10,240 bytes is presented to a member of the *exec* family.

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*(4)).

9 EBADF Bad file number

Either a file descriptor refers to no open file, or a read (respectively, write) request is made to a file which is open only for writing (respectively, reading).

10 ECHILD No child processes

A *wait* was executed by a process that had no existing or unwaited-for child processes.

11 EAGAIN No more processes

A *fork* failed because the system's process table is full or the user is not allowed to create any more processes.

12 ENOMEM Not enough space

During an *exec*, *brk*, or *sbrk*, a program asks for more space than the system is able to supply. The maximum allocation is 3.5 megabytes; a program that gets this condition with a smaller allocation may work at another time when other large programs aren't hogging the swap file. If this problem recurs, the system administrator may want to consider enlarging the swap file.

The error may also occur if the arrangement of text, data, and stack segments requires too many 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 in a way forbidden by the protection system. From *locking*, an attempt to lock bytes already under a checking lock.

14 EFAULT Bad address

The system encountered a hardware fault in attempting to use an argument of a system call.

15 ENOTBLK Block device required

A non-block file was mentioned where a block device was required, e.g., in *mount*.

16 EBUSY Device or resource busy

An attempt was made to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment). It will also occur if an attempt is made to enable accounting when it is already enabled. The device or resource is currently unavailable.

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 prefix or as an argument to *chdir*(2).

21 EISDIR Is a directory

An attempt was made to write on a directory.

22 EINVAL Invalid argument

Some invalid argument (e.g., dismounting a non-mounted device; mentioning an undefined signal in *signal*, or *kill*; reading or writing a file for which *lseek* has generated a negative pointer). Also set by the math functions described in the (3M) entries of this manual.

23 ENFILE File table overflow

The system file table is full, and temporarily no more *opens* can be accepted.

24 EMFILE Too many open files

No process may have more than 20 file descriptors open at a time.

25 ENOTTY Not a character device

An attempt was made to *ioctl*(2) a file that is not a special character device.

26 ETXTBSY Text file busy

An attempt was made to execute a pure-procedure program that is currently open for writing. 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 file size (1,082,201,088 bytes) or ULIMIT; see *ulimit*(2).

28 ENOSPC No space left on device
During a *write* to an ordinary file, there is no free space left on the device. This can occur on a PILF file when the file system lacks unallocated clusters as big as the file's cluster size. On System 6600 tape files, it indicates a read past the end of the tape.

29 ESPIPE Illegal seek
An *lseek* was issued to a pipe.

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 the maximum number of links (1000) 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 not representable within machine precision.

35 ENOMSG No message of desired type
An attempt was made to receive a message of a type that does not exist on the specified message queue; see *msgop*(2).

36 EIDRM Identifier Removed
This error is returned to processes that resume execution due to the removal of an identifier from the file system's name space (see *msgctl*(2), *semctl*(2), and *shmctl*(2)).

50 EBADE Invalid exchange
Use of an invalid Inter-CPU Communication exchange descriptor.

51 EBADR Invalid request descriptor
Use of an invalid Inter-CPU Communication request descriptor.

52 EXFULL Exchange full
An Inter-CPU Communication request failed because an exchange is full. The exchange might be the request's response exchange or the service exchange.

53 ENOANO No anode
The Application Processor has as many files open as it can handle.

54 EBADRQC Invalid request code
No operating system or RTOS process is servicing the specified request code.

56 EDEADLOCK Deadlock error
Call cannot be honored because of potential deadlock or because lock table is full. See *locking*(2).

DEFINITIONS

Process ID
Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 1 to 30,000.

Parent Process ID

A new process is created by a currently active process; see *fork*(2). The parent process ID of a process is the process ID of its creator.

Process Group ID

Each active process is a member of a process group that is identified by a positive integer called the process group ID. This ID is the process ID of the group leader. This grouping permits the signaling of related processes; see *kill*(2).

Tty Group ID

Each active process can be a member of a terminal group that is identified by a positive integer called the tty group ID. This grouping is used to terminate a group of related processes upon termination of one of the processes in the group; see *exit*(2) and *signal*(2).

Real User ID and Real Group ID

Each user allowed on the system is identified by a positive integer called a real user ID.

Each user is also a member of a group. The group is identified by a positive integer called the real group ID.

An active process has a real user ID and real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for the creation of the process.

Effective User ID and Effective Group ID

An active process has an effective user ID and an effective group ID that are used to determine file access permissions (see below). The effective user ID and effective group ID are equal to the process's real user ID and real group ID respectively, unless the process or one of its ancestors evolved from a file that had the set-user-ID bit or set-group ID bit set; see *exec*(2).

Super-user

A process is recognized as a *super-user* process and is granted special privileges if its effective user ID is 0.

Special Processes

The processes with a process ID of 0 and a process ID of 1 are special processes and are referred to as *proc0* and *proc1*.

Proc0 is the scheduler. *Proc1* is the initialization process (*init*). Proc1 is the ancestor of every other process in the system and is used to control the process structure.

File Descriptor

A file descriptor is a small integer used to do I/O on a file. The value of a file descriptor is from 0 to 19. A process may have no more than 20 file descriptors (0-19) open simultaneously. A file descriptor is returned by system calls such as *open*(2), or *pipe*(2). The file descriptor is used as an argument by calls such as *read*(2), *write*(2), *ioctl*(2), and *close*(2).

File Name

Names consisting of 1 to 14 characters may be used to name an ordinary file, special file or directory.

These characters may be selected from the set of all character values excluding \0 (null) and the ASCII code for / (slash).

Note that it is generally unwise to use *, ?, [, or] as part of file names because of the special meaning attached to these characters by the shell. See *sh*(1). Although permitted, it is advisable to avoid the use of unprintable characters in file names.

Path Name and Path Prefix

A path name is a null-terminated character string starting with an optional slash (/), followed by zero or more directory names separated by slashes, optionally followed by a file name.

More precisely, a path name is a null-terminated character string constructed as follows:

<path-name>::=<file-name>| <path-prefix><file-name>|/
<path-prefix>::=<rtprefix>| /<rtprefix>
<rtprefix>::=<dirname>/| <rtprefix><dirname>/

where <file-name> is a string of 1 to 14 characters other than the ASCII slash and null, and <dirname> is a string of 1 to 14 characters (other than the ASCII slash and null) that names a directory.

If a path name begins with a slash, the path search begins at the *root* directory. Otherwise, the search begins from the current working directory.

A slash by itself names the root directory.

Unless specifically stated otherwise, the null path name is treated as if it named a non-existent file.

Directory

Directory entries are called links. By convention, a directory contains at least two links, . and .., referred to as *dot* and *dot-dot* respectively. Dot refers to the directory itself and dot-dot refers to its parent directory.

Root Directory and Current Working Directory

Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving path name searches. The root directory of a process need not be the root directory of the root file system.

File Access Permissions

Read, write, and execute/search permissions on a file are granted to a process if one or more of the following are true:

The effective user ID of the process is super-user.

The effective user ID of the process matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion (0700) of the file mode is set.

The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process matches the group of the file and the appropriate access bit of the "group" portion (070) of the file mode is set.

The effective user ID of the process does not match the user ID of the owner of the file, and the effective group ID of the process does not match the group ID of the file, and the appropriate access bit of the "other" portion (07) of the file mode is set.

Otherwise, the corresponding permissions are denied.

Message Queue Identifier

A message queue identifier (msqid) is a unique positive integer created by a *msgget*(2) system call. Each msqid has a message queue and a data structure associated with it. The data structure is referred to as *msqid_ds* and contains the following members:

```
struct   ipc_perm msg_perm;  /* operation permission struct */
ushort   msg_qnum;           /* number of msgs on q */
ushort   msg_qbytes;         /* max number of bytes on q */
ushort   msg_lspid;          /* pid of last msgsnd operation */
ushort   msg_lrpid;          /* pid of last msgrcv operation */
time_t   msg_stime;          /* last msgsnd time */
time_t   msg_rtime;          /* last msgrcv time */
time_t   msg_ctime;          /* last change time */
                             /* Times measured in secs since */
                             /* 00:00:00 GMT, Jan. 1, 1970 */
```

Msg_perm is an ipc_perm structure that specifies the message operation permission (see below). This structure includes the following members:

```
ushort   cuid;      /* creator user id */
ushort   cgid;      /* creator group id */
ushort   uid;       /* user id */
ushort   gid;       /* group id */
ushort   mode;      /* r/w permission */
```

Msg_qnum is the number of messages currently on the queue. **Msg_qbytes** is the maximum number of bytes allowed on the queue. **Msg_lspid** is the process id of the last process that performed a *msgsnd* operation. **Msg_lrpid** is the process id of the last process that performed a *msgrcv* operation. **Msg_stime** is the time of the last *msgsnd* operation, **msg_rtime** is the time of the last *msgrcv* operation, and **msg_ctime** is the time of the last *msgctl*(2) operation that changed a member of the above structure.

Message Operation Permissions

In the *msgop*(2) and *msgctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

00400	Read by user
00200	Write by user
00060	Read, Write by group
00006	Read, Write by others

Read and Write permissions on a msqid are granted to a process if one or more of the following are true:

The effective user ID of the process is super-user.

The effective user ID of the process matches **msg_perm.[c]uid** in the data structure associated with *msqid* and the appropriate bit of the "user" portion (0600) of **msg_perm.mode** is set.

The effective user ID of the process does not match **msg_perm.[c]uid** and the effective group ID of the process matches **msg_perm.[c]gid** and the appropriate bit of the "group" portion (060) of **msg_perm.mode** is set.

The effective user ID of the process does not match **msg_perm.[c]uid** and the effective group ID of the process does not match **msg_perm.[c]gid** and the appropriate bit of the "other" portion (06) of **msg_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Semaphore Identifier

A semaphore identifier (semid) is a unique positive integer created by a *semget*(2) system call. Each semid has a set of semaphores and a data structure associated with it. The data structure is referred to as *semid_ds* and contains the following members:

```
struct   ipc_perm sem_perm;  /* operation permission struct */
ushort   sem_nsems;          /* number of sems in set */
time_t   sem_otime;          /* last operation time */
time_t   sem_ctime;          /* last change time */
                             /* Times measured in secs since */
                             /* 00:00:00 GMT, Jan. 1, 1970 */
```

Sem_perm is an ipc_perm structure that specifies the semaphore operation permission (see below). This structure includes the following members:

```
ushort   cuid;      /* creator user id */
ushort   cgid;      /* creator group id */
ushort   uid;       /* user id */
ushort   gid;       /* group id */
ushort   mode;      /* r/a permission */
```

The value of **sem_nsems** is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a positive integer referred to as a *sem_num*. Sem_num values run sequentially from 0 to the value of sem_nsems minus 1. **Sem_otime** is the time of the last *semop*(2) operation, and **sem_ctime** is the time of the last *semctl*(2) operation that changed a member of the above structure.

A semaphore is a data structure that contains the following members:

```
ushort   semval;     /* semaphore value */
short    sempid;     /* pid of last operation  */
ushort   semncnt;    /* # awaiting semval > cval */
ushort   semzcnt;    /* # awaiting semval = 0 */
```

Semval is a non-negative integer. **Sempid** is equal to the process ID of the last process that performed a semaphore operation on this semaphore. **Semncnt** is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become greater than its current value. **Semzcnt** is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become zero.

Semaphore Operation Permissions

In the *semop*(2) and *semctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

00400	Read by user
00200	Alter by user
00060	Read, Alter by group
00006	Read, Alter by others

Read and Alter permissions on a semid are granted to a process if one or more of the following are true:

The effective user ID of the process is super-user.

The effective user ID of the process matches **sem_perm.[c]uid** in the data structure associated with *semid* and the appropriate bit of the "user" portion (0600) of **sem_perm.mode** is set.

The effective user ID of the process does not match **sem_perm.[c]uid** and the effective group ID of the process matches **sem_perm.[c]gid** and the appropriate bit of the "group" portion (060) of **sem_perm.mode** is set.

The effective user ID of the process does not match **sem_perm.[c]uid** and the effective group ID of the process does not match **sem_perm.[c]gid** and the appropriate bit of the "other" portion (06) of **sem_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

Shared Memory Identifier

A shared memory identifier (shmid) is a unique positive integer created by a *shmget*(2) system call. Each shmid has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. The data structure is referred to as *shmid_ds* and contains the following members:

```
struct   ipc_perm shm_perm;  /* operation permission struct */
int      shm_segsz;          /* size of segment */
ushort   shm_cpid;           /* creator pid */
ushort   shm_lpid;           /* pid of last operation */
short    shm_nattch;         /* number of current attaches */
time_t   shm_atime;          /* last attach time */
time_t   shm_dtime;          /* last detach time */
time_t   shm_ctime;          /* last change time */
                             /* Times measured in secs since */
```

```
                                        /* 00:00:00 GMT, Jan. 1, 1970 */
```

Shm_perm is an ipc_perm structure that specifies the shared memory operation permission (see below). This structure includes the following members:

```
ushort   cuid;      /* creator user id */
ushort   cgid;      /* creator group id */
ushort   uid;       /* user id */
ushort   gid;       /* group id */
ushort   mode;      /* r/w permission */
```

Shm_segsz specifies the size of the shared memory segment. **Shm_cpid** is the process id of the process that created the shared memory identifier. **Shm_lpid** is the process id of the last process that performed a *shmop*(2) operation. **Shm_nattch** is the number of processes that currently have this segment attached. **Shm_atime** is the time of the last *shmat* operation, **shm_dtime** is the time of the last *shmdt* operation, and **shm_ctime** is the time of the last *shmctl*(2) operation that changed one of the members of the above structure.

Shared Memory Operation Permissions

In the *shmop*(2) and *shmctl*(2) system call descriptions, the permission required for an operation is given as "{token}", where "token" is the type of permission needed interpreted as follows:

00400	Read by user
00200	Write by user
00060	Read, Write by group
00006	Read, Write by others

Read and Write permissions on a shmid are granted to a process if one or more of the following are true:

The effective user ID of the process is super-user.

The effective user ID of the process matches **shm_perm.[c]uid** in the data structure associated with *shmid* and the appropriate bit of the "user" portion (0600) of **shm_perm.mode** is set.

The effective user ID of the process does not match **shm_perm.[c]uid** and the effective group ID of the process matches **shm_perm.[c]gid** and the appropriate bit of the "group" portion (060) of **shm_perm.mode** is set.

The effective user ID of the process does not match **shm_perm.[c]uid** and the effective group ID of the process does not match **shm_perm.[c]gid** and the appropriate bit of the "other" portion (06) of **shm_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

SEE ALSO

close(2), ioctl(2), open(2), pipe(2), read(2), write(2), intro(3).

NAME

access – determine accessibility of a file

SYNOPSIS

int access (path, amode)
char *path;
int amode;

DESCRIPTION

Path points to a path name naming a file. *Access* checks the named file for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID. The bit pattern contained in *amode* is constructed as follows:

04	read
02	write
01	execute (search)
00	check existence of file

Access to the file is denied if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	Read, write, or execute (search) permission is requested for a null path name.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[EROFS]	Write access is requested for a file on a read-only file system.
[ETXTBSY]	Write access is requested for a pure procedure (shared text) file that is being executed.
[EACCESS]	Permission bits of the file mode do not permit the requested access.
[EFAULT]	*Path* points outside the allocated address space for the process.

The owner of a file has permission checked with respect to the "owner" read, write, and execute mode bits, members of the file's group other than the owner have permissions checked with respect to the "group" mode bits, and all others have permissions checked with respect to the "other" mode bits.

RETURN VALUE

If the requested access is permitted, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), stat(2).

NAME

acct – enable or disable process accounting

SYNOPSIS

int acct (path)
char *path;

DESCRIPTION

Acct is used to enable or disable the system process accounting routine. If the routine is enabled, an accounting record will be written on an accounting file for each process that terminates. Termination can be caused by one of two things: an *exit* call or a signal; see *exit*(2) and *signal*(2). The effective user ID of the calling process must be super-user to use this call.

Path points to a path name naming the accounting file. The accounting file format is given in *acct*(4).

The accounting routine is enabled if *path* is non-zero and no errors occur during the system call. It is disabled if *path* is zero and no errors occur during the system call.

Acct will fail if one or more of the following are true:

[EPERM]	The effective user of the calling process is not super-user.
[EBUSY]	An attempt is being made to enable accounting when it is already enabled.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	One or more components of the accounting file path name do not exist.
[EACCES]	A component of the path prefix denies search permission.
[EACCES]	The file named by *path* is not an ordinary file.
[EACCES]	*Mode* permission is denied for the named accounting file.
[EISDIR]	The named file is a directory.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*Path* points to an illegal address.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

exit(2), signal(2), acct(4).

NAME

alarm – set a process alarm clock

SYNOPSIS

unsigned alarm (sec)
unsigned sec;

DESCRIPTION

Alarm instructs the alarm clock of the calling process to send the signal **SIGALRM** to the calling process after the number of real time seconds specified by *sec* have elapsed; see *signal*(2).

Alarm requests are not stacked; successive calls reset the alarm clock of the calling process.

If *sec* is 0, any previously made alarm request is canceled.

RETURN VALUE

Alarm returns the amount of time previously remaining in the alarm clock of the calling process.

SEE ALSO

pause(2), signal(2).

NAME

brk, sbrk – change data segment space allocation

SYNOPSIS

int brk (endds)
char *endds;

char *sbrk (incr)
int incr;

DESCRIPTION

Brk and *sbrk* are used to change dynamically the amount of space allocated for the calling process's data segment; see *exec*(2). The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. The newly allocated space is set to zero.

Brk sets the break value to *endds* and changes the allocated space accordingly.

Sbrk adds *incr* bytes to the break value and changes the allocated space accordingly. *Incr* can be negative, in which case the amount of allocated space is decreased.

Brk and *sbrk* will fail without making any change in the allocated space if one or more of the following are true:

Such a change would result in the process exceeding its allocation limit. This can be imposed by the system administrator (see *ulimit (2)*); otherwise it is the available space is the processor's swap file, with an absolute maximum of about 3.5 megabytes. [ENOMEM]

Such a change would result in the break value being greater than or equal to the start address of any attached shared memory segment (see *shmop*(2)).

RETURN VALUE

Upon successful completion, *brk* returns a value of 0 and *sbrk* returns the old break value. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2).

NAME

chdir – change working directory

SYNOPSIS

int chdir (path)
char *path;

DESCRIPTION

Path points to the path name of a directory. *Chdir* causes the named directory to become the current working directory, the starting point for path searches for path names not beginning with /.

Chdir will fail and the current working directory will be unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path name is not a directory.
[ENOENT]	The named directory does not exist.
[EACCES]	Search permission is denied for any component of the path name.
[EFAULT]	*Path* points outside the allocated address space of the process.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

chroot(2).

NAME

chmod – change mode of file

SYNOPSIS

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

DESCRIPTION

Path points to a path name naming a file. *Chmod* sets the access permission portion of the named file's mode according to the bit pattern contained in *mode*.

Access permission bits are interpreted as follows:

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 if a directory) by owner.
00070	Read, write, execute (search) by group.
00007	Read, write, execute (search) by others.

The effective user ID of the process must match the owner of the file or be super-user to change the mode of a file.

If the effective user ID of the process is not super-user, mode bit 01000 (save text image on execution) is cleared.

If the effective user ID of the process is not super-user and the effective group ID of the process does not match the group ID of the file, mode bit 02000 (set group ID on execution) is cleared.

If an executable file is prepared for sharing then mode bit 01000 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.

Chmod will fail and the file mode will be unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not super-user.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*Path* points outside the allocated address space of the process.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

chown(2), mknod(2).

NAME

chown – change owner and group of a file

SYNOPSIS

int chown (path, owner, group)
char *path;
int owner, group;

DESCRIPTION

Path points to a path name naming a file. The owner ID and group ID of the named file are set to the numeric values contained in *owner* and *group* respectively.

Only processes with effective user ID equal to the file owner or super-user may change the ownership of a file.

If *chown* is invoked by other than the super-user, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000 respectively, will be cleared.

Chown will fail and the owner and group of the named file will remain unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not super-user.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*Path* points outside the allocated address space of the process.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

chown(1), chmod(2).

NAME

chroot – change root directory

SYNOPSIS

int chroot (path)
char *path;

DESCRIPTION

Path points to a path name naming a directory. *Chroot* causes the named directory to become the root directory, the starting point for path searches for path names beginning with /. The user's working directory is unaffected by the *chroot* system call.

The effective user ID of the process must be super-user to change the root directory.

The .. entry in the root directory is interpreted to mean the root directory itself. Thus, .. cannot be used to access files outside the subtree rooted at the root directory.

Chroot will fail and the root directory will remain unchanged if one or more of the following are true:

[ENOTDIR]	Any component of the path name is not a directory.
[ENOENT]	The named directory does not exist.
[EPERM]	The effective user ID is not super-user.
[EFAULT]	*Path* points outside the allocated address space of the process.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

chdir(2).

NAME

close – close a file descriptor

SYNOPSIS

int close (fildes)
int fildes;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Close* closes the file descriptor indicated by *fildes*.

[EBADF] *Close* will fail if *fildes* is not a valid open file descriptor.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), exec(2), fcntl(2), open(2), pipe(2).

NAME

creat – create a new file or rewrite an existing one

SYNOPSIS

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

DESCRIPTION

Creat creates a new ordinary file or prepares to rewrite an existing file named by the path name pointed to by *path*.

If the file exists, the length is truncated to 0 and the mode and owner are unchanged; if a PILF file, the cluster size exponent is also unchanged. Otherwise, the file's owner ID is set to the effective user ID, of the process the group ID of the process is set to the effective group ID, of the process and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows:

All bits set in the process's file mode creation mask are cleared. See *umask*(2).

The "save text image after execution bit" of the mode is cleared. See *chmod*(2).

The process's default cluster size exponent determines the cluster size of files created on PILF file systems. See *syslocal*(2).

Upon successful completion, the file descriptor is returned and the file is open for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across *exec* system calls. See *fcntl*(2). No process may have more than 20 files open simultaneously. A new file may be created with a mode that forbids writing.

Creat will fail if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[ENOENT]	The path name is null.
[EACCES]	The file does not exist and the directory in which the file is to be created does not permit writing.
[EROFS]	The named file resides or would reside on a read-only file system.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed.
[EACCES]	The file exists and write permission is denied.
[EISDIR]	The named file is an existing directory.
[EMFILE]	Twenty (20) file descriptors are currently open.
[EFAULT]	*Path* points outside the allocated address space of the process.
[ENFILE]	The system file table is full.
[EDEADLOCK]	A side effect of a previous *locking*(2) call.

RETURN VALUE

Upon successful completion, a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), close(2), dup(2), fcntl(2), locking(2), lseek(2), open(2), read(2), umask(2), write(2).

NAME

dup – duplicate an open file descriptor

SYNOPSIS

int dup (fildes)
int fildes;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Dup* returns a new file descriptor having the following in common with the original:

Same open file (or pipe).

Same file pointer (i.e., both file descriptors share one file pointer).

Same access mode (read, write or read/write).

The new file descriptor is set to remain open across *exec* system calls. See *fcntl*(2).

The file descriptor returned is the lowest one available.

Dup will fail if one or more of the following are true:

[EBADF] *Fildes* is not a valid open file descriptor.

[EMFILE] Twenty (20) file descriptors are currently open.

RETURN VALUE

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), close(2), exec(2), fcntl(2), open(2), pipe(2).

NAME

exCall – Send a request and wait for the response

SYNOPSIS

#include <exch.h>

exCall(reqbl);
struct reqheader *reqbl;

DESCRIPTION

ExCall sends a request and waits for the response. *Reqbl* must point to a request block that describes the message. The request block has four parts: a request header, control information, request PbCbs, and response PbCbs.

The ICC user include file defines a request header thus:

```
struct rqheader {
        unsigned short r_sCntInfo;
        unsigned char r_nReqPbCb;
        unsigned char r_nRespPbCb;
        unsigned short r_userNum;
        unsigned short r_exchResp;
        unsigned short r_ercRet;
        unsigned short r_rqCode;
};
```

The client sets the following fields: *r_sCntInfo* (which must be even), *r_nReqPbCb*, and *r_nRespPbCb*, specify the size of the rest of the request block; *r_exchResp*, specifies where the response must be sent; and *r_rqCode*, specifies the destination of the request. The kernel and server ignore any values in *r_userNum* or *r_ercRet*. Each request code requires specific values for *r_sCntInfo*, *r_nReqPbCb*, and *r_nRespPbCb*.

The client uses the control information to send fixed-length data fields to the server.

A PbCb has the following structure:

```
struct PbCb {
        char *pc_offset;
        unsigned short pc_count;
};
```

The client uses Request PbCbs to send blocks of data to the server. Each PbCb gives the location (*pc_offset*) and size (*pc_count*) of a data block.

The client uses Response PbCbs to pass response data areas (*pc_offset*) and maximum lengths (*pc_count*) to the server and kernel. If the server ignores the restrictions, the kernel right-truncates the offending fields.

The memory containing the variable-length fields need not immediately follow the request block.

SEE ALSO

Operating System Programmer's Guide, Section 22.

RETURN VALUE

–1 indicates error, with an error code in *errno*. See *perror*(3).

WARNINGS

If the service is provided by RTOS, integer data must have Intel byte ordering. See *shortswap*(3).

Lint(1) may complain that *exCall* argument types are inconsistent, especially if the client uses more than one kind of request block. To suppress these complaints, cast the argument to its official type:

```
exCall((struct rqheader *) reqbl);
```

Use of this cast does not affect the object code.

NAME

exQueryDfltRespExch, exAllocExch, exDeallocExch – obtain and abandon exchanges

SYNOPSIS

```
#include <exch.h>

unsigned char exQueryDfltRespExch();

unsigned char exAllocExch();

exDeallocExch(ex)
unsigned char ex;
```

DESCRIPTION

A process that wants to receive messages must own exchanges. Each exchange has an exchange descriptor, unique only to the exchange's owner.

ExQueryDfltRespExch returns the descriptor of the caller's default response exchange. Every process has a default response exchange as soon as it is forked. A process must reference its default response exchange explicitly. A process can use its default response exchange to receive both requests and responses.

ExAllocExch allocates a new exchange and returns its exchange descriptor. The calling process can use this exchange to receive both requests and responses.

ExDeallocExch deallocates the specified exchange. Any requests still waiting or on their way to the exchange are rejected with a return code of **0xFF**. Any responses still waiting or on their way to the exchange are discarded.

A process's death deallocates all its exchanges, but an *exec* has no affect on exchanges.

SEE ALSO

Operating System Programmer's Guide, Section 22.

RETURN VALUE

–1 indicates error, with an error code in *errno*. See *perror*(3).

NAME

exCpRequest, exReject – remove a request from an exchange

SYNOPSIS

```
#include <exch.h>

exCpRequest(reqdes, reqst)
unsigned short reqdes;
struct rqheader *reqst;

exReject(reqdes, r_ercRet)
unsigned short reqdes;
unsigned short r_ercRet;
```

DESCRIPTION

ExCpRequest and *exReject* both remove a request from a server's exchange. A server that wants to examine the request uses *exCpRequest*; a server that has no interest in the messages's contents uses *exReject*.

ExCpRequest copies the message indicated by the request describtor, *reqdes*. The kernel places the request block and request data blocks together at the location pointed to by *reqst*. *Reqst* must be an even address; each data block appears at an even address. (The amount of memory the message requires is returned by a check on the message queue; see *exWait*(2I).) The kernel sets the request PbCbs to point to the server's copies of the data blocks.

ExReject discards the contents of the indicated message. It sends the response, with the return code (*m_ercRet* in the request block header) set to *r_ercRet*.

FILES

/usr/include/exch.h – ICC user include file

SEE ALSO

Operating System Programmer's Guide, Section 22.

RETURN VALUE

–1 indicates error, with an error code in *errno*. See *perror*(3).

NAME

exCpResponse, exDiscard – remove a response from an exchange

SYNOPSIS

```
#include <exch.h>

exCpResponse(reqdes, reqst)
unsigned char reqdes;
struct rqheader *reqst;

exDiscard(reqdes)
unsigned char reqdes;
```

DESCRIPTION

ExCpResponse and *exDiscard* both remove a response from an exchange. A client that wants to examine the response uses *exCpResponse;* a client that has no interest in the message's contents uses *exDiscard.*

ExCpResponse copies the message indicated by the request descriptor *reqdes*. The kernel uses the request block pointed to by *reqst* to place the parts of the response:

- The error code goes in the *r_ercRet* field of the request block header.
- The kernel examines each response PbCb in the request block. The *pc_offset* field should be set to the location reserved for the data; *pc_count* should be set to the number of bytes available at that location. If the server provided more than *pc_count* bytes, the kernel right-truncates the data to fit. The kernel overwrites *pc_count* with the number of bytes actually transferred.

ExDiscard discards the contents of the indicated message. It returns the message's return code field (*m_ercRet* in the request block header).

FILES

/usr/include/exch.h – ICC user include file

SEE ALSO

Operating System Programmer's Guide, Section 22.

RETURN VALUE

-1 indicates error, with an error code in *errno*. See *perror*(3).

WARNINGS

If the service is provided by RTOS, integer data must have Intel byte ordering. See *shortswap*(3).

NAME

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

SYNOPSIS

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

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

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

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

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

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

DESCRIPTION

Exec in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary, executable file called the *new process file*. This file consists of a header (see *a.out*(4)), a text segment, and a data segment. The data segment contains an initialized portion and an uninitialized portion (bss). There can be no return from a successful *exec* because the calling process is overlaid by the new process.

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.

Path points to a path name that identifies the new process file.

File points to the new process file. The path prefix for this file is obtained by a search of the directories passed as the *environment* line "PATH =" (see *environ*(5)). The environment is supplied by the shell (see *sh*(1)).

Arg0, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or its last component).

Argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or its last component). *Argv* is terminated by a null pointer.

Envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process. *Envp* is terminated by a null pointer. For *execl* and *execv*, the C run-time start-off routine places a pointer to the environment of the calling process in the global cell:

```
extern char **environ;
```

and it is used to pass the environment of the calling process to the new process.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; see *fcntl*(2). For those file descriptors that remain open, the file pointer

is unchanged.

Signals set to terminate the calling process will be set to terminate the new process. Signals set to be ignored by the calling process will be set to be ignored by the new process. Signals set to be caught by the calling process will be set to terminate the new process; see *signal*(2).

If the set-user-ID mode bit of the new process file is set (see *chmod*(2)), *exec* sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID and real group ID of the new process remain the same as those of the calling process.

The shared memory segments attached to the calling process will not be attached to the new process (see *shmop*(2)).

Profiling is disabled for the new process; see *profil*(2).

The new process also inherits the following attributes from the calling process:

nice value (see *nice*(2))
process ID
parent process ID
process group ID
ICC exchanges, together with unremoved messages addressed to them
semadj values (see *semop*(2))
tty group ID (see *exit*(2) and *signal*(2))
trace flag (see *ptrace*(2) request 0)
time left until an alarm clock signal (see *alarm*(2))
current working directory
root directory
file mode creation mask (see *umask*(2))
file size limit (see *ulimit*(2))
utime, *stime*, *cutime*, and *cstime* (see *times*(2))
PILF cluster size exponent for this process

Exec will fail and return to the calling process if one or more of the following are true:

[ENOENT] One or more components of the new process path name of the file do not exist.

[ENOTDIR] A component of the new process path of the file prefix is not a directory.

[EACCES] Search permission is denied for a directory listed in the new process file's path prefix.

[EACCES] The new process file is not an ordinary file.

[EACCES] The new process file mode denies execution permission.

[ENOEXEC] The exec is not an *execlp* or *execvp*, and the new process file has the appropriate access permission but an invalid magic number in its header.

[ETXTBSY] The new process file is a pure procedure (shared text) file that is currently open for writing by some process.

[ENOMEM] The new process requires more memory than is allowed by the system-imposed maximum MAXMEM.

[E2BIG] The number of bytes in the new process's argument list is greater than the system-imposed limit of 10,240 bytes.

[EFAULT] The new process file is not as long as indicated by the size values in its header.

[EFAULT] *Path*, *argv*, or *envp* point to an illegal address.

RETURN VALUE

If *exec* returns to the calling process an error has occurred; the return value will be –1 and *errno* will be set to indicate the error.

SEE ALSO

sh(1), alarm(2), exit(2), fork(2), nice(2), ptrace(2), semop(2), signal(2), times(2), ulimit(2), umask(2), a.out(4), environ(5).

NAME

exSendOnDealloc, exCnxSendOnDealloc – make final requests

SYNOPSIS

#include <exch.h>

unsigned char exSendOnDealloc(reqblk)
struct rqheader *reqblk;

exCnxSendOnDealloc(req)
unsigned char req;

DESCRIPTION

ExSendOnDealloc specifies a request and returns a request descriptor in precisely the same manner as *exRequest*. But where *exRequest* dispatches the request immediately, *exSendOnDealloc* puts a hold on the request. When the client process deallocates the request's response exchange, either by dying or by a call to *exDealloc* (see *exchanges*(2)), the kernel delivers the message.

ExCnxSendOnDealloc cancels the specified message. *req* must be a value returned by a call to *exSendOnDealloc*.

FILES

/usr/include/exch.h – ICC user include file

SEE ALSO

Operating System Programmer's Guide, Section 22.

RETURN VALUE

–1 indicates error, with an error code in *errno*. See *perror*(3).

WARNINGS

The server must respond to the message, even though there's no one to read the response.

NAME

exit, _exit – terminate process

SYNOPSIS

void exit (status)
int status;
void _exit (status)
int status;

DESCRIPTION

Exit terminates the calling process with the following consequences:

All of the file descriptors open in the calling process are closed.

If the parent process of the calling process is executing a *wait*, it is notified of the calling process's termination and the low order eight bits (i.e., bits 0377) of *status* are made available to it; see *wait*(2).

If the parent process of the calling process is not executing a *wait*, the calling process is transformed into a zombie process. A *zombie process* is a process that only occupies a slot in the process table. It has no other space allocated either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information (see <**sys/proc.h**>) to be used by *times*.

The parent process ID of all of the calling process's existing child processes and zombie processes is set to 1. This means the initialization process (see *intro*(2)) inherits each of these processes.

All ICC exchanges are deallocated. (Process termination is the only way to deallocate the default response exchange.)

Each attached shared memory segment is detached and the value of **shm_nattach** in the data structure associated with its shared memory identifier is decremented by 1.

For each semaphore for which the calling process has set a semadj value (see *semop*(2)), that semadj value is added to the semval of the specified semaphore.

An accounting record is written on the accounting file if the system's accounting routine is enabled; see *acct*(2).

If the process ID, tty group ID, and process group ID of the calling process are equal, the SIGHUP signal is sent to each process that has a process group ID equal to that of the calling process.

The C function *exit* may cause cleanup actions before the process exits. The function *_exit* circumvents all cleanup.

SEE ALSO

acct(2), intro(2), exchanges(2), semop(2), signal(2), wait(2).

WARNING

See *WARNING* in *signal*(2).

NAME

exRequest – Send a message to a server

SYNOPSIS

#include <exch.h>

unsigned char exRequest(reqbl);
struct reqheader *reqbl;

DESCRIPTION

ExRequest sends a message to a server. *reqbl* must point to a request block that describes the message. *exRequest* returns a request descriptor; this descriptor appears in subsequent references to the request by the client or the kernel.

The request block has four parts: a request header, control information, request PbCbs, and response PbCbs.

A request header has the following structure.

```
struct rqheader {
        unsigned short r_sCntInfo;
        unsigned char r_nReqPbCb;
        unsigned char r_nRespPbCb;
        unsigned short r_userNum;
        unsigned short r_exchResp;
        unsigned short r_ercRet;
        unsigned short r_rqCode;
};
```

The client sets the following fields: *r_sCntInfo*(which *must* be *even)*, *r_nReqPbCb*, and *r_nRespPbCb*, specify the size of the rest of the request block; *r_exchResp*, specifies where the response must be sent; and *r_rqCode*, specifies the destination of the request. The kernel and server ignore any values in *r_userNum* or *r_ercRet*. Each request code requires specific values for *r_sCntInfo*, *r_nReqPbCb*, and *r_nRespPbCb*.

The client uses the control information to send fixed-length data fields to the server.

A PbCb has the following structure:

```
struct PbCb {
        char *pc_offset;
        unsigned short pc_count;
};
```

The client uses Request PbCbs to send request data blocks to the server. Each PbCb gives the location (*pc_offset*) and size (*pc_count*) of a data block.

The client uses Response PbCbs to pass response data-length restrictions to the server. The client sets the *pc_count* field of each response PbCb to the maximum length for that data block.

The locations containing the client's request data need not immediately follow the request block.

The kernel copies the complete message immediately: once *exRequest* returns, it is safe to modify the message.

After the client has sent the request, it must watch for the corresponding response (*exWait*(2)) and specify the response's disposition (*exCpResponse*(2)).

SEE ALSO

Operating System Programmer's Guide, Section 22.

RETURN VALUE

–1 indicates error, with an error code in *errno*. See *perror*(3).

WARNINGS

Use of *exRequest* requires more client-kernel interaction than is necessary for most requests. Compare *exCall*(1).

If the service is provided by RTOS, integer data has Intel byte ordering. See *shortswap*(3).

lint(1) may complain that *exRequest* argument types are inconsistent, especially if the client uses more than one kind of request block. To suppress these complaints, cast the argument to its official type:

```
exRequest((struct rqheader *) reqbl);
```

Use of this cast does not affect the object code.

NAME

exRespond – send a message to a client

SYNOPSIS

#include <exch.h>

exRespond(reqdes, reqbl)
unsigned char reqdes;
struct rqheader *reqbl;

DESCRIPTION

ExRespond issues a response to a specific request. The request descriptor *reqdes* specifies that request. *reqbl* points to a request block that describes the response. This request block has the same format as the request block that described the request (see *exRequest*(2)). The server only sets the error return code fields and each of the response PbCbs.

The kernel copies the complete message immediately: once *exRespond* returns, it is safe to modify the message.

The memory containing the server's variable-length response fields need not directly follow the request block.

SEE ALSO

Operating System Programmer's Guide, Section 22.

RETURN VALUE

–1 indicates error, with an error code in *errno*. See *perror*(3).

NAME

exServeRq – appropriate a request code

SYNOPSIS

#include <exch.h>

exServeRq(exch, code);
unsigned char exch;
unsigned short code;

DESCRIPTION

A server (a process that receives requests) must own a request code for use by clients (processes that send requests). *exServeRq* appropriates *code* as a request code and assigns the request to the exchange specified by *exch*. If *exch* is zero, the process gives up *code*, which can then be appropriated by another server.

Any process can appropriate a request code, but only one can own it at a time.

Codes 0 through 0xBFFF (49151) are reserved for Motorola system services. Each installation should reserve additional codes for local system services. User services must not use reserved codes, even if they do not currently identify a service.

SEE ALSO

Operating System Programmer's Guide, Section 22.

RETURN VALUE

–1 indicates error, with an error code in *errno*. See *perror*(3).

NAME

exWait, exCheck – examine an ICC message queue

SYNOPSIS

```
#include <exch.h>

exCheck(ex, mstat);
unsigned char ex;
struct msgret *mstat;

exWait(ex, mstat);
unsigned char ex;
struct msgret *mstat;
```

DESCRIPTION

Each call to *exWait* or *exCheck* returns with information on the oldest unnoticed message waiting at the exchange whose descriptor is *ex*. An unnoticed message is one that *exWait* and *exCheck* have not reported on since the last time a message was removed from the exchange. When an exchange's owner removes a message, all messages still waiting become "unnoticed" again; see *excpresponse*(2) and *excprequest*(2). *Excall*(2) never affects the "noticed" status of any message.

ExWait and *exCheck* write a report to the memory pointed to by *mstat* . The report has the following structure:

```
struct msgret {
        unsigned short m_rqCode;
        unsigned short m_reqdes;
        int m_size;
        char m_flag;
        unsigned short m_ercRet;
        unsigned char m_cputype;
        unsigned char m_slot;
        struct request *m_offset;
};
```

When the process takes further action on this message (copying it from the message queue; if it's a request, sending a response) it passes the kernel *m_reqdes* to identify the specific message.

exWait and *exCheck* differ only in their "no messages" action. If no unnoticed messages wait at the specified exchange, *exWait* waits for a new one to arrive; *exCheck* returns immediately with an error code.

The calling process must still specify some action on each message. See *excpresponse*(2) and *excprequest*(2).

SEE ALSO

Operating System Programmer's Guide, Section 22.

RETURN VALUE

Error returns –1 with an error code in *errno*. See *perror*(3).

NAME

fcntl – file control

SYNOPSIS

#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd, arg;

DESCRIPTION

Fcntl provides for control over open files. *Fildes* is an open file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

The *commands* available are:

F_DUPFD — Return a new file descriptor as follows:

Lowest numbered available file descriptor greater than or equal to *arg*.

Same open file (or pipe) as the original file.

Same file pointer as the original file (i.e., both file descriptors share one file pointer).

Same access mode (read, write or read/write).

Same file status flags (i.e., both file descriptors share the same file status flags).

The close-on-exec flag associated with the new file descriptor is set to remain open across *exec*(2) system calls.

F_GETFD — Get the close-on-exec flag associated with the file descriptor *fildes*. If the low-order bit is **0** the file will remain open across *exec*, otherwise the file will be closed upon execution of *exec*.

F_SETFD — Set the close-on-exec flag associated with *fildes* to the low-order bit of *arg* (**0** or **1** as above).

F_GETFL — Get *file* status flags.

F_SETFL — Set *file* status flags to *arg*. Only certain flags can be set; see *fcntl*(5).

Fcntl will fail if one or more of the following are true:

[EBADF] — *Fildes* is not a valid open file descriptor.

[EMFILE] — *Cmd* is F_DUPFD and 20 file descriptors are currently open.

[EMFILE] — *Cmd* is F_DUPFD and *arg* is negative, greater than 20, or greater than the largest unallocated descriptor.

RETURN VALUE

Upon successful completion, the value returned depends on *cmd* as follows:

F_DUPFD	A new file descriptor.
F_GETFD	Value of flag (only the low-order bit is defined).
F_SETFD	Value other than –1.
F_GETFL	Value of file flags.
F_SETFL	Value other than –1.

Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

close(2), exec(2), open(2), fcntl(5).

NAME

fork – create a new process

SYNOPSIS

int fork ( )

DESCRIPTION

Fork causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). This means the child process inherits the following attributes from the parent process:

environment
close-on-exec flag (see *exec*(2))
signal handling settings (i.e., **SIG_DFL**, **SIG_ING**, function address)
set-user-ID mode bit
set-group-ID mode bit
profiling on/off status
nice value (see *nice*(2))
all attached shared memory segments (see *shmop*(2))
process group ID
tty group ID (see *exit*(2) and *signal*(2))
trace flag (see *ptrace*(2) request 0)
time left until an alarm clock signal (see *alarm*(2))
current working directory
root directory
file mode creation mask (see *umask*(2))
file size limit (see *ulimit*(2))
PILF cluster size exponent (System 6600 only; see *pilf*(5)).

The child process differs from the parent process in the following ways:

The child process has a unique process ID.

The child process has a different parent process ID (i.e., the process ID of the parent process).

The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

All semadj values are cleared (see *semop*(2)).

The child process's *utime*, *stime*, *cutime*, and *cstime* are set to 0. The time left until an alarm clock signal is reset to 0.

On System 6600 systems, the child inherits no Inter-CPU Communication exchanges from the parent. Initially, the child's only exchange is the default response exchange.

Fork will fail and no child process will be created if one or more of the following are true:

[EAGAIN] The system-imposed limit on the total number of processes under execution would be exceeded.

[EAGAIN] The system-imposed limit on the total number of processes under execution by a single user would be exceeded.

RETURN VALUE

Upon successful completion, *fork* returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of –1 is returned to the parent process, no child process is created, and *errno* is set to indicate the error.

SEE ALSO

exchanges(2), exec(2), nice(2), plock(2), ptrace(2), semop(2), shmop(2), signal(2), times(2), ulimit(2), umask(2), wait(2).

NAME

getpid, getpgrp, getppid – get process, process group, and parent process IDs

SYNOPSIS

int getpid ( )

int getpgrp ( )

int getppid ( )

DESCRIPTION

Getpid returns the process ID of the calling process.

Getpgrp returns the process group ID of the calling process.

Getppid returns the parent process ID of the calling process.

SEE ALSO

exec(2), fork(2), intro(2), setpgrp(2), signal(2).

NAME

getuid, geteuid, getgid, getegid – get real user, effective user, real group, and effective group IDs

SYNOPSIS

unsigned short getuid ( )

unsigned short geteuid ( )

unsigned short getgid ( )

unsigned short getegid ( )

DESCRIPTION

Getuid returns the real user ID of the calling process.

Geteuid returns the effective user ID of the calling process.

Getgid returns the real group ID of the calling process.

Getegid returns the effective group ID of the calling process.

SEE ALSO

intro(2), setuid(2).

NAME

ioctl – control device

SYNOPSIS

ioctl (fildes, request, arg)
int fildes, request;

DESCRIPTION

Ioctl performs a variety of functions on character special files (devices). The write-ups of various devices in Section 7 discuss how *ioctl* applies to them.

Ioctl will fail if one or more of the following are true:

[EBADF]	*Fildes* is not a valid open file descriptor.
[ENOTTY]	*Fildes* is not associated with a character special device.
[EINVAL]	*Request* or *arg* is not valid. See Section 7.
[EINTR]	A signal was caught during the *ioctl* system call.

RETURN VALUE

If an error has occurred, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

termio(7).

NAME

kill – send a signal to a process or a group of processes

SYNOPSIS

int kill (pid, sig)
int pid, sig;

DESCRIPTION

Kill sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal that is to be sent is specified by *sig* and is either one from the list given in *signal*(2), or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The real or effective user ID of the sending process must match the real or effective user ID of the receiving process, unless the effective user ID of the sending process is super-user.

The processes with a process ID of 0 and a process ID of 1 are special processes (see *intro*(2)) and will be referred to below as *proc0* and *proc1*, respectively.

If *pid* is greater than zero, *sig* will be sent to the process whose process ID is equal to *pid*. *Pid* may equal 1.

If *pid* is 0, *sig* will be sent to all processes excluding *proc0* and *proc1* whose process group ID is equal to the process group ID of the sender.

If *pid* is –1 and the effective user ID of the sender is not super-user, *sig* will be sent to all processes excluding *proc0* and *proc1* whose real user ID is equal to the effective user ID of the sender.

If *pid* is –1 and the effective user ID of the sender is super-user, *sig* will be sent to all processes excluding *proc0* and *proc1*.

If *pid* is negative but not –1, *sig* will be sent to all processes whose process group ID is equal to the absolute value of *pid*.

Kill will fail and no signal will be sent if one or more of the following are true:

[EINVAL]	*Sig* is not a valid signal number.
[EINVAL]	*Sig* is SIGKILL and *pid* is 1 (proc1).
[ESRCH]	No process can be found corresponding to that specified by *pid*.
[EPERM]	The user ID of the sending process is not super-user, and its real or effective user ID does not match the real or effective user ID of the receiving process.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

kill(1), getpid(2), setpgrp(2), signal(2).

NAME

link – link to a file

SYNOPSIS

int link (path1, path2)
char *path1, *path2;

DESCRIPTION

Path1 points to a path name naming an existing file. *Path2* points to a path name naming the new directory entry to be created. *Link* creates a new link (directory entry) for the existing file.

Link will fail and no link will be created if one or more of the following are true:

[ENOTDIR]	A component of either path prefix is not a directory.
[ENOENT]	A component of either path prefix does not exist.
[EACCES]	A component of either path prefix denies search permission.
[ENOENT]	The file named by *path1* does not exist.
[EEXIST]	The link named by *path2* exists.
[EPERM]	The file named by *path1* is a directory and the effective user ID is not super-user.
[EXDEV]	The link named by *path2* and the file named by *path1* are on different logical devices (file systems).
[ENOENT]	*Path2* points to a null path name.
[EACCES]	The requested link requires writing in a directory with a mode that denies write permission.
[EROFS]	The requested link requires writing in a directory on a read-only file system.
[EFAULT]	*Path* points outside the allocated address space of the process.
[EMLINK]	The maximum number of links to a file would be exceeded.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

unlink(2).

NAME

locking – exclusive access to regions of a file

SYNOPSIS

int locking (filedes, mode, size);
int fildes, mode;
long size;

DESCRIPTION

Locking places or removes a kernel-enforced lock on a region of a file. The calling process has exclusive access to regions it has locked. If another process uses *read*(2), *write*(2), *creat*(2), or *open*(2) (with O_TRUNC) in a way that reads or modifies part of the locked region, the second process's system call does not return until the lock is released, unless deadlock or some other error is detected. A process whose execution is suspended in such a manner is said to be *blocked*.

Parameters specify the file to be locked or unlocked, the kind of lock or unlock, and the region affected:

- *Filedes* specifies the file to be locked or unlocked; *filedes* is a file descriptor returned by an *open*, *create*, *pipe*, *fcntl*, or *dup* system call.
- *Mode* specifies the action: 0 for lock removal; 1 for blocking lock; 2 for checking lock. Blocking and checking locks differ only if the attempted lock is itself locked out: a blocking lock waits until the existing lock or locks are removed; a checking lock immediately returns an error.
- The region affected begins at the current file offset associated with *filedes* and is *size* bytes long. If *size* is zero, the region affected ends at the end of the file.

Locking imposes no structure on an operating system file. A process can arbitrarily lock any unlocked byte and unlock any locked byte. However, creating a large number of noncontiguous locked regions can fill up the system's lock table and make further locks impossible. It is advisable that a program's use of *locking* segment the file in the same way as does the program's use of *read* and *write*.

A process is said to be deadlocked if it is sleeping until an unlocking which is indirectly prevented by that same sleeping process. The kernel will not permit a *read*, *write*, *creat*, *open* with O_TRUNC, or blocking *locking* if such a call would deadlock the calling process. *Errno* is set to EDEADLOCK. The standard response to such a situation is for the program to release all its existing locked areas and try again. If a *locking* call fails because the kernel's table of locked areas is full, again, *errno* is set to EDEADLOCK and, again, the calling program should release its existing locked areas.

Special files and pipes can be locked, but no input/output is blocked.

Locks are automatically removed if the process that placed the lock terminates or closes the file descriptor used to place the lock.

SEE ALSO

create(2), close(2), dup(2), open(2), read(2), write(2).

RETURN VALUE

A return value of *-1* indicates an error, with the error value in *errno*.

[EACCES] A checking lock on a region already locked.

[EDEADLOCK] A lock that would cause deadlock or overflow the system's lock table.

WARNING

Do not apply any standard input/output library function to a locked file: this library does not know about *locking*.

NAME

lseek – move read/write file pointer

SYNOPSIS

long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;

DESCRIPTION

Fildes is a file descriptor returned from a *creat*, *open*, *dup*, or *fcntl* system call. *Lseek* sets the file pointer associated with *fildes* 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*.

Upon successful completion, the resulting pointer location, as measured in bytes from the beginning of the file, is returned.

Lseek will fail and the file pointer will remain unchanged if one or more of the following are true:

[EBADF] *Fildes* is not an open file descriptor.

[ESPIPE] *Fildes* is associated with a pipe or fifo.

[EINVAL and SIGSYS signal]
Whence is not 0, 1, or 2.

[EINVAL] The resulting file pointer would be negative.

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

RETURN VALUE

Upon successful completion, a non-negative integer indicating the file pointer value is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), fcntl(2), open(2).

NAME

mknod – make a directory, or a special or ordinary file

SYNOPSIS

int mknod (path, mode, dev)
char *path;
int mode, dev;

DESCRIPTION

Mknod creates a new file named by the path name pointed to by *path*. The mode of the new file is initialized from *mode*. Where the value of *mode* is interpreted as follows:

0170000 file type; one of the following:
- 0010000 fifo special
- 0020000 character special
- 0040000 directory
- 0060000 block special
- 0100000 or 0000000 ordinary file

0004000 set user ID on execution
0002000 set group ID on execution
0001000 save text image after execution
0000777 access permissions; constructed from the following
- 0000400 read by owner
- 0000200 write by owner
- 0000100 execute (search on directory) by owner
- 0000070 read, write, execute (search) by group
- 0000007 read, write, execute (search) by others

The owner ID of the file is set to the effective user ID of the process. The group ID of the file is set to the effective group ID of the process.

Values of *mode* other than those above are undefined and should not be used. The low-order 9 bits of *mode* are modified by the process's file mode creation mask: all bits set in the process's file mode creation mask are cleared. See *umask*(2). If *mode* indicates a block or character special file, *dev* is a configuration-dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored.

Mknod may be invoked only by the super-user for file types other than FIFO special.

Mknod will fail and the new file will not be created if one or more of the following are true:

[EPERM]	The effective user ID of the process is not super-user.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of the path prefix does not exist.
[EROFS]	The directory in which the file is to be created is located on a read-only file system.
[EEXIST]	The named file exists.
[EFAULT]	*Path* points outside the allocated address space of the process.

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

mkdir(1), chmod(2), exec(2), umask(2), fs(4).

NAME

mount – mount a file system

SYNOPSIS

int mount (spec, dir, rwflag)
char *spec, *dir;
int rwflag;

DESCRIPTION

Mount requests that a removable file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. *Spec* and *dir* are pointers to path names.

Upon successful completion, references to the file *dir* will refer to the root directory on the mounted file system.

The low-order bit of *rwflag* is used to control write permission on the mounted file system; if **1**, writing is forbidden, otherwise writing is permitted according to individual file accessibility.

Mount may be invoked only by the super-user.

Mount will fail if one or more of the following are true:

[EPERM]	The effective user ID is not super-user.
[ENOENT]	Any of the named files does not exist.
[ENOTDIR]	A component of a path prefix is not a directory.
[ENOTBLK]	*Spec* is not a block special device.
[ENXIO]	The device associated with *spec* does not exist.
[ENOTDIR]	*Dir* is not a directory.
[EFAULT]	*Spec* or *dir* points outside the allocated address space of the process.
[EBUSY]	*Dir* is currently mounted on, is someone's current working directory, or is otherwise busy.
[EBUSY]	The device associated with *spec* is currently mounted.
[EBUSY]	There are no more mount table entries.
[EROFS]	The low-order bit of *rwflag* is zero and the volume containing the file system is physically write-protected.

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

umount(2).

NAME

msgctl – message control operations

SYNOPSIS

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

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;
```

DESCRIPTION

Msgctl provides a variety of message control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT — Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET — Set the value of the following members of the data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode /* only low 9 bits */
msg_qbytes
```

This *cmd* can only be executed by a process that has an effective user ID equal to either that of super user or to the value of **msg_perm.uid** in the data structure associated with *msqid*. Only super user can raise the value of **msg_qbytes**.

IPC_RMID — Remove the message queue identifier specified by *msqid* from the system and destroy the message queue and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of super user or to the value of **msg_perm.uid** in the data structure associated with *msqid*.

Msgctl will fail if one or more of the following are true:

[EINVAL] — *Msqid* is not a valid message queue identifier.

[EINVAL] — *Cmd* is not a valid command.

[EACCES] — *Cmd* is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process (see *intro*(2)).

[EPERM] — *Cmd* is equal to **IPC_RMID** or **IPC_SET**. The effective user ID of the calling process is not equal to that of super user and it is not equal to the value of **msg_perm.uid** in the data structure associated with *msqid*.

[EPERM] — *Cmd* is equal to **IPC_SET,** an attempt is being made to increase to the value of **msg_qbytes,** and the effective user ID of the calling process is not equal to that of super user.

[EFAULT] — *Buf* points to an illegal address.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

intro(2), msgget(2), msgop(2).

NAME

msgget – get message queue

SYNOPSIS

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

int msgget (key, msgflg)
key_t key;
int msgflg;
```

DESCRIPTION

Msgget returns the message queue identifier associated with *key*.

A message queue identifier and associated message queue and data structure (see *intro*(2)) are created for *key* if one of the following are true:

10 *Key* is equal to **IPC_PRIVATE**.

Key does not already have a message queue identifier associated with it, and (*msgflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

Msg_perm.cuid, **msg_perm.uid**, **msg_perm.cgid**, and **msg_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of **msg_perm.mode** are set equal to the low-order 9 bits of *msgflg*.

Msg_qnum, **msg_lspid**, **msg_lrpid**, **msg_stime**, and **msg_rtime** are set equal to 0.

Msg_ctime is set equal to the current time.

Msg_qbytes is set equal to the system limit.

Msgget will fail if one or more of the following are true:

[EACCES] A message queue identifier exists for *key*, but operation permission (see *intro*(2)) as specified by the low-order 9 bits of *msgflg* would not be granted.

[ENOENT] A message queue identifier does not exist for *key* and (*msgflg* & **IPC_CREAT**) is "false".

[ENOSPC] A message queue identifier is to be created but the system-imposed limit on the maximum number of allowed message queue identifiers system wide would be exceeded.

[EEXIST] A message queue identifier exists for *key* but ((*msgflg* & IPC_CREAT) & (*msgflg* & **IPC_EXCL**)) is "true".

RETURN VALUE

Upon successful completion, a non-negative integer, namely a message queue identifier, is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

intro(2), msgctl(2), msgop(2).

NAME

msgop – message operations

SYNOPSIS

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

int msgsnd (msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz, msgflg;

int msgrcv (msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

DESCRIPTION

Msgsnd is used to send a message to the queue associated with the message queue identifier specified by *msqid*. {WRITE} *Msgp* points to a structure containing the message. This structure is composed of the following members:

```
long    mtype;      /* message type */
char    mtext[];    /* message text */
```

Mtype is a positive integer that can be used by the receiving process for message selection (see *msgrcv* below). *Mtext* is any text of length *msgsz* bytes. *Msgsz* can range from 0 to a system-imposed maximum.

Msgflg specifies the action to be taken if one or more of the following are true:

The number of bytes already on the queue is equal to **msg_qbytes** (see *intro*(2)).

The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

If (*msgflg* & IPC_NOWAIT) is "true", the message will not be sent and the calling process will return immediately.

If (*msgflg* & IPC_NOWAIT) is "false", the calling process will suspend execution until one of the following occurs:

The condition responsible for the suspension no longer exists, in which case the message is sent.

Msqid is removed from the system (see *msgctl*(2)). When this occurs, *errno* is set equal to EIDRM, and a value of –1 is returned.

The calling process receives a signal that is to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in *signal*(2)).

Msgsnd will fail and no message will be sent if one or more of the following are true:

[EINVAL] *Msqid* is not a valid message queue identifier.

[EACCES] Operation permission is denied to the calling process (see *intro*(2)).

[EINVAL] *Mtype* is less than 1.

[EAGAIN] The message cannot be sent for one of the reasons cited above and (*msgflg* & **IPC_NOWAIT**) is "true".

[EINVAL] *Msgsz* is less than zero or greater than the system-imposed limit.

[EFAULT] *Msgp* points to an illegal address.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* (see intro (2)).

Msg_qnum is incremented by 1.

Msg_lspid is set equal to the process ID of the calling process.

Msg_stime is set equal to the current time.

Msgrcv reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the structure pointed to by *msgp*. {READ} This structure is composed of the following members:

```
long    mtype;      /* message type */
char    mtext[];    /* message text */
```

Mtype is the received message's type as specified by the sending process. *Mtext* is the text of the message. *Msgsz* specifies the size in bytes of *mtext*. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & **MSG_NOERROR**) is "true". The truncated part of the message is lost and no indication of the truncation is given to the calling process.

Msgtyp specifies the type of message requested as follows:

If *msgtyp* is equal to 0, the first message on the queue is received.

If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.

If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

Msgflg specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

If (*msgflg* & **IPC_NOWAIT**) is "true", the calling process will return immediately with a return value of –1 and *errno* set to ENOMSG.

If (*msgflg* & **IPC_NOWAIT**) is "false", the calling process will suspend execution until one of the following occurs:

A message of the desired type is placed on the queue.

Msqid is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of –1 is returned.

The calling process receives a signal that is to be caught. In this case a message is not received and the calling process resumes execution in the manner prescribed in *signal*(2)).

Msgrcv will fail and no message will be received if one or more of the following are true:

[EINVAL] *Msqid* is not a valid message queue identifier.

[EACCES] Operation permission is denied to the calling process.

[EINVAL] *Msgsz* is less than 0.

[E2BIG] Mtext is greater than *msgsz* and (*msgflg* & **MSG_NOERROR**) is "false".

[ENOMSG] The queue does not contain a message of the desired type and (*msgtyp* & **IPC_NOWAIT**) is "true".

[EFAULT] *Msgp* points to an illegal address.

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* (see intro (2)).

Msg_qnum is decremented by 1.

Msg_lrpid is set equal to the process ID of the calling process.

Msg_rtime is set equal to the current time.

RETURN VALUES

If *msgsnd* or *msgrcv* return due to the receipt of a signal, a value of –1 is returned to the calling process and *errno* is set to EINTR. If they return due to removal of *msqid* from the system, a value of –1 is returned and *errno* is set to EIDRM.

Upon successful completion, the return value is as follows:

Msgsnd returns a value of 0.

Msgrcv returns a value equal to the number of bytes actually placed into *mtext*.

Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

intro(2), msgctl(2), msgget(2), signal(2).

NAME

nice – change priority of a process

SYNOPSIS

int nice (incr)
int incr;

DESCRIPTION

Nice adds the value of *incr* to the nice value of the calling process. A process's *nice value* is a positive number for which a more positive value results in lower CPU priority.

A maximum nice value of 39 and a minimum nice value of 0 are imposed by the system. Requests for values above or below these limits result in the nice value being set to the corresponding limit.

[EPERM] *Nice* will fail and not change the nice value if *incr* is negative or greater than 40 and the effective user ID of the calling process is not super-user.

RETURN VALUE

Upon successful completion, *nice* returns the new nice value minus 20. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

nice(1), exec(2).

NAME

open – open for reading or writing

SYNOPSIS

```
#include <fcntl.h>
int open (path, oflag [ , mode ] )
char *path;
int oflag, mode;
```

DESCRIPTION

Path points to a path name naming a file. *Open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. *Oflag* values are constructed by or-ing flags from the following list (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY
Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY This flag may affect subsequent reads and writes. See *read*(2) and *write*(2).

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NDELAY is set:

An *open* for reading-only will return without delay. An *open* for writing-only will return an error if no process currently has the file open for reading.

If O_NDELAY is clear:

An *open* for reading-only will block until a process opens the file for writing. An *open* for writing-only will block until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:

The open will return without waiting for carrier.

If O_NDELAY is clear:

The open will block until carrier is present.

O_APPEND If set, the file pointer will be set to the end of the file prior to each write.

O_CREAT If the file exists, this flag has no effect. Otherwise, the owner ID of the file is set to the effective user ID of the process, the group ID of the file is set to the effective group ID of the process, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows (see *creat*(2)):

All bits set in the file mode creation mask of the process are cleared. See *umask*(2).

The "save text image after execution bit" of the mode is cleared. See *chmod*(2).

The process's default cluster size exponent determines the cluster size of files created on PILF file systems.

O_TRUNC If the file exists, its length is truncated to 0 and the mode and owner are unchanged.

O_EXCL If O_EXCL and O_CREAT are set, *open* will fail if the file exists.

O_DIRECT (MegaFrame only.) I/O is direct between the process's address space and the disk, bypassing the kernel's buffer cache. See *pilf*(5).

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across *exec* system calls. See *fcntl*(2).

The named file is opened unless one or more of the following are true:

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] O_CREAT is not set and the named file does not exist.

[EACCES] A component of the path prefix denies search permission.

[EACCES] *Oflag* permission is denied for the named file.

[EISDIR] The named file is a directory and *oflag* is write or read/write.

[EROFS] The named file resides on a read-only file system and *oflag* is write or read/write.

[EMFILE] Twenty (20) file descriptors are currently open.

[ENXIO] The named file is a character special or block special file, and the device associated with this special file does not exist.

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write.

[EFAULT] *Path* points outside the allocated address space of the process.

[EEXIST] O_CREAT and O_EXCL are set, and the named file exists.

[ENXIO] O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.

[EINTR] A signal was caught during the *open* system call.

[ENFILE] The system file table is full.

[EDEADLOCK] A side effect of a previous *locking*(2) call, when applying O_TRUNC .

RETURN VALUE

Upon successful completion, the file descriptor is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), close(2), creat(2), dup(2), fcntl(2), locking(2), lseek(2), read(2), umask(2), write(2), pilf(5).

NAME

openi – open a file specified by i-node

SYNOPSIS

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

int openi (dev, inode, oflag)
dev_t dev;
ino_t inode;
int oflag;
```

DESCRIPTION

Openi permits access to a file without reference to any of its directory links. Because it doesn't use the directory hierarchy, *openi* doesn't require any access permission except from the file itself. Use of *openi* must be authorized in advance by *syslocal*(2).

Dev specifies the device number of the file system that contains the file. *Inode* is the i-number of the file. *Oflag* is a set of open flags, identical to those used with *open*(2). The return value is a file descriptor, like that returned by *open*.

A file descriptor returned by *openi* has the same properties as one returned by *open*. It counts against the per-process limit of 20 file descriptors.

The specified file is opened unless one or more of the following are true:

The specified inode is not allocated. [ENOENT]

Oflag permission is denied for the named file. [EACCES]

The named file is a directory. [EISDIR]

The named file resides on a read-only file system and *oflag* is write or read/write. [EROFS]

Twenty (20) file descriptors are currently open. [EMFILE]

The named file is a character special or block special file. [ENXIO]

The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write. [ETXTBSY]

Path points outside the process's allocated address space. [EFAULT]

O_CREAT and O_EXCL are set, and the named file exists. [EEXIST]

O_NDELAY is set, the file is a FIFO, O_WRONLY is set, and no process has the file open for reading. [ENXIO]

The specified file system is not mounted. [ENXIO]

RETURN VALUE

On success, returns a file descriptor, a nonnegative integer. On failure, returns –1 and sets *errno*.

SEE ALSO

creat(2), open(2), syslocal(2).

NAME

pause – suspend process until signal

SYNOPSIS

pause ( )

DESCRIPTION

Pause suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process.

If the signal causes termination of the calling process, *pause* will not return.

If the signal is *caught* by the calling process and control is returned from the signal-catching function (see *signal*(2)), the calling process resumes execution from the point of suspension; with a return value of –1 from *pause* and *errno* set to EINTR.

SEE ALSO

alarm(2), kill(2), signal(2), wait(2).

NAME

pipe – create an interprocess channel

SYNOPSIS

int pipe (fildes)
int fildes[2];

DESCRIPTION

Pipe creates an I/O mechanism called a pipe and returns two file descriptors, *fildes*[0] and *fildes*[1]. *Fildes*[0] is opened for reading and *fildes*[1] is opened for writing.

Up to 5120 bytes of data are buffered by the pipe before the writing process is blocked. A read only file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out (FIFO) basis.

[EMFILE]	*Pipe* will fail if 19 or more file descriptors are currently open.
[ENFILE]	The system file table is full.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

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

NAME

profil – execution time profile

SYNOPSIS

void 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 (octal) gives a 1-1 mapping of pc's to words in *buff*; 077777 (octal) maps each pair of instruction words together. 02(octal) 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 will be turned off if an update in *buff* would cause a memory fault.

RETURN VALUE

Not defined.

SEE ALSO

prof(1), monitor(3C).

NAME

ptrace – process trace

SYNOPSIS

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

DESCRIPTION

Ptrace provides a means by which a parent process may control the execution of a child process. Its primary use is for the implementation of breakpoint debugging; see *sdb*(1). The child process behaves normally until it encounters a signal (see *signal*(2) for the list), at which time it enters a stopped state and its parent is notified via *wait*(2). When the child is in the stopped state, its parent can examine and modify its "core image" using *ptrace*. Also, the parent can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

The *request* argument determines the precise action to be taken by *ptrace* and is one of the following:

0 This request must be issued by the child process if it is to be traced by its parent. It turns on the child's trace flag that stipulates that the child should be left in a stopped state upon receipt of a signal rather than the state specified by *func*; see *signal*(2). The *pid*, *addr*, and *data* arguments are ignored, and a return value is not defined for this request. Peculiar results will ensue if the parent does not expect to trace the child.

The remainder of the requests can only be used by the parent process. For each, *pid* is the process ID of the child. The child must be in a stopped state before these requests are made.

1, 2 With these requests, the word at location *addr* in the address space of the child is returned to the parent process. If I and D space are separated (as on PDP-11s), request **1** returns a word from I space, and request **2** returns a word from D space. If I and D space are not separated (as on Motorola Series 6000-family processors, the 3B 20S computer, and VAX-11/780), either request 1 or request 2 may be used with equal results. The *data* argument is ignored. These two requests will fail if *addr* is not the start address of a word, in which case a value of –1 is returned to the parent process and the parent's *errno* is set to EIO.

3 With this request, the word at location *addr* in the child's USER area in the system's address space (see **<sys/user.h>**) is returned to the parent process. Addresses in this area range from 0 to 8192 on Motorola Series 6000-family processors, 0 to 1024 on the PDP-11s and 0 to 2048 on the 3B 20 computer and VAX. The *data* argument is ignored. This request will fail if *addr* is not the start address of a word or is outside the USER area, in which case a value of –1 is returned to the parent process and the parent's *errno* is set to EIO.

4, 5 With these requests, the value given by the *data* argument is written into the address space of the child at location *addr*. If I and D space are separated (as on PDP-11s), request 4 writes a word into I space, and request 5 writes a word into D space. If I and D space are not separated (as on Motorola Series 6000-family processors, the 3B 20 computer, and VAX), either request 4 or request 5 may be used with equal results. Upon successful completion, the value written into the address space of the child is returned to the parent. These two requests will fail if *addr* is a location in a pure procedure space and another process is executing in that space, or *addr* is not the start address of a word. Upon failure a value of –1 is returned to the parent process and the parent's *errno* is set to EIO.

6 With this request, a few entries in the child's USER area can be written. *Data* gives the value that is to be written and *addr* is the location of the entry. The few

entries that can be written are:

the general registers (i.e., registers 0-15 on Motorola Series 6000-family processors, registers 0-11 on the 3B 20S computer, registers 0-7 on PDP-11s, and registers 0-15 on the VAX)

the condition codes of the Processor Status Word on the 3B 20 computer

the floating point status register and six floating point registers on PDP-11s

certain bits of the Processor Status Word on PDP-11s (i.e, bits 0–4, and 8–11)

certain bits of the Processor Status Longword on the VAX (i.e., bits 0–7, 16–20, and 30–31).

Motorola Series 6000-family processors: all processor status bits except 8, 9, 10, and 13.

7 This request causes the child to resume execution. If the *data* argument is 0, all pending signals including the one that caused the child to stop are canceled before it resumes execution. If the *data* argument is a valid signal number, the child resumes execution as if it had incurred that signal, and any other pending signals are canceled. The *addr* argument must be equal to 1 for this request. Upon successful completion, the value of *data* is returned to the parent. This request will fail if *data* is not 0 or a valid signal number, in which case a value of –1 is returned to the parent process and the parent's *errno* is set to EIO.

8 This request causes the child to terminate with the same consequences as *exit*(2).

9 This request sets the trace bit in the Processor Status Word of the child (i.e., bit 15 on Motorola Series 6000-family processors, bit 4 on PDP-11s; bit 30 on the VAX) and then executes the same steps as listed above for request **7**. The trace bit causes an interrupt upon completion of one machine instruction. This effectively allows single stepping of the child. On the 3B 20S computer there is no trace bit and this request returns an error.
Note: the trace bit remains set after an interrupt on PDP-11s but is turned off after an interrupt on the VAX.

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**.

GENERAL ERRORS

Ptrace will in general fail if one or more of the following are true:

[EIO] *Request* is an illegal number.

[ESRCH] *Pid* identifies a child that does not exist or has not executed a *ptrace* with request **0**.

SEE ALSO

exec(2), signal(2), wait(2).

NAME

read – read from file

SYNOPSIS

int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

Read attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*.

On devices capable of seeking, the *read* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *read*, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line (see *ioctl*(2) and *termio*(7)), or if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

When attempting to read from an empty pipe (or FIFO):

If O_NDELAY is set, the read will return a 0.

If O_NDELAY is clear, the read will block until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a tty that has no data currently available:

If O_NDELAY is set, the read will return a 0.

If O_NDELAY is clear, the read will block until data becomes available.

Read will fail if one or more of the following are true:

[EBADF]	*Fildes* is not a valid file descriptor open for reading.
[EFAULT]	*Buf* points outside the allocated address space.
[EINTR]	A signal was caught during the *read* system call.
[EDEADLOCK]	A side effect of a previous *locking*(2) call.

RETURN VALUE

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. Otherwise, a –1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), fcntl(2), ioctl(2), locking(2), open(2), pipe(2), termio(7).

NAME

semctl – semaphore control operations

SYNOPSIS

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

int semctl (semid, semnum, cmd, arg)
int semid, cmd;
int semnum;
union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
} arg;
```

DESCRIPTION

Semctl provides a variety of semaphore control operations as specified by *cmd*.

The following *cmd*s are executed with respect to the semaphore specified by *semid* and *semnum:*

GETVAL — Return the value of semval (see *intro*(2)). {READ}

SETVAL — Set the value of semval to *arg.val*. {ALTER} When this cmd is successfully executed, the semadj value corresponding to the specified semaphore in all processes is cleared.

GETPID — Return the value of sempid. {READ}

GETNCNT — Return the value of semncnt. {READ}

GETZCNT — Return the value of semzcnt. {READ}

The following *cmd*s return and set, respectively, every semval in the set of semaphores.

GETALL — Place semvals into array pointed to by *arg.array*. {READ}

SETALL — Set semvals according to the array pointed to by *arg.array*. {ALTER} When this cmd is successfully executed the semadj values corresponding to each specified semaphore in all processes are cleared.

The following *cmd*s are also available:

IPC_STAT — Place the current value of each member of the data structure associated with *semid* into the structure pointed to by *arg.buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET — Set the value of the following members of the data structure associated with *semid* to the corresponding value found in the structure pointed to by *arg.buf*:

sem_perm.uid
sem_perm.gid
sem_perm.mode /* only low 9 bits */

This cmd can only be executed by a process that has an effective user ID equal to either that of super-user or to the value of **sem_perm.uid** in the data structure associated with *semid*.

IPC_RMID — Remove the semaphore identifier specified by *semid* from the system and destroy the set of semaphores and data structure associated with it. This cmd can only be executed by a process that has an effective user ID equal to either that of super-user or to the value of **sem_perm.uid** in the data structure associated with *semid*.

Semctl will fail if one or more of the following are true:

[EINVAL]	*Semid* is not a valid semaphore identifier.
[EINVAL]	*Semnum* is less than zero or greater than **sem_nsems**.
[EINVAL]	*Cmd* is not a valid command.
[EACCES]	Operation permission is denied to the calling process (see *intro*(2)).
[ERANGE]	*Cmd* is **SETVAL** or **SETALL** and the value to which semval is to be set is greater than the system imposed maximum.
[EPERM]	*Cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of super-user and it is not equal to the value of **sem_perm.uid** in the data structure associated with *semid*.
[EFAULT]	*Arg.buf* points to an illegal address.

RETURN VALUE

Upon successful completion, the value returned depends on *cmd* as follows:

GETVAL	The value of semval.
GETPID	The value of sempid.
GETNCNT	The value of semncnt.
GETZCNT	The value of semzcnt.
All others	A value of 0.

Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

intro(2), semget(2), semop(2).

NAME

semget – get set of semaphores

SYNOPSIS

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

int semget (key, nsems, semflg)
key_t key;
int nsems, semflg;
```

DESCRIPTION

Semget returns the semaphore identifier associated with *key*.

A semaphore identifier and associated data structure and set containing *nsems* semaphores (see *intro*(2)) are created for *key* if one of the following are true:

Key is equal to **IPC_PRIVATE**.

Key does not already have a semaphore identifier associated with it, and (*semflg* & **IPC_CREAT**) is "true".

Upon creation, the data structure associated with the new semaphore identifier is initialized as follows:

Sem_perm.cuid, **sem_perm.uid**, **sem_perm.cgid**, and **sem_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of **sem_perm.mode** are set equal to the low-order 9 bits of *semflg*.

Sem_nsems is set equal to the value of *nsems*.

Sem_otime is set equal to 0 and **sem_ctime** is set equal to the current time.

Semget will fail if one or more of the following are true:

[EINVAL] *Nsems* is either less than or equal to zero or greater than the system-imposed limit.

[EACCES] A semaphore identifier exists for *key*, but operation permission (see *intro*(2)) as specified by the low-order 9 bits of *semflg* would not be granted.

[EINVAL] A semaphore identifier exists for *key*, but the number of semaphores in the set associated with it is less than *nsems* and *nsems* is not equal to zero.

[ENOENT] A semaphore identifier does not exist for *key* and (*semflg* & **IPC_CREAT**) is "false".

[ENOSPC] A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphore identifiers system wide would be exceeded.

[ENOSPC] A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores system wide would be exceeded.

[EEXIST] A semaphore identifier exists for *key* but ((*semflg* & IPC_CREAT) **and** (*semflg* & **IPC_EXCL**)) is "true".

RETURN VALUE

Upon successful completion, a non-negative integer, namely a semaphore identifier, is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

intro(2), semctl(2), semop(2).

NAME

semop – semaphore operations

SYNOPSIS

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

int semop (semid, sops, nsops)
int semid;
struct sembuf **sops;
int nsops;
```

DESCRIPTION

Semop is used to atomically perform an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by *semid*. *Sops* is a pointer to the array of semaphore-operation structures. *Nsops* is the number of such structures in the array. The contents of each structure includes the following members:

```
short   sem_num;   /* semaphore number */
short   sem_op;    /* semaphore operation */
short   sem_flg;   /* operation flags */
```

Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore specified by *semid* and *sem_num*.

Sem_op specifies one of three semaphore operations as follows:

If *sem_op* is a negative integer, one of the following will occur: {ALTER}

If semval (see *intro*(2)) is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from semval. Also, if (*sem_flg* & SEM_UNDO) is "true", the absolute value of *sem_op* is added to the calling process's semadj value (see *exit*(2)) for the specified semaphore. All processes suspended waiting for semval are rescheduled.

If semval is less than the absolute value of *sem_op* and (*sem_flg* & IPC_NOWAIT) is "true", *semop* will return immediately.

If semval is less than the absolute value of *sem_op* and (*sem_flg* & IPC_NOWAIT) is "false", *semop* will increment the semncnt associated with the specified semaphore and suspend execution of the calling process until one of the following conditions occur.

Semval becomes greater than or equal to the absolute value of *sem_op*. When this occurs, the value of semncnt associated with the specified semaphore is decremented, the absolute value of *sem_op* is subtracted from semval and, if (*sem_flg* & SEM_UNDO) is "true", the absolute value of *sem_op* is added to the calling process's semadj value for the specified semaphore, and all the operations are tried again.

The semid for which the calling process is awaiting action is removed from the system (see *semctl*(2)). When this occurs, *errno* is set equal to EIDRM, and a value of –1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of semncnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(2).

If *sem_op* is a positive integer, the value of *sem_op* is added to semval and, if (*sem_flg* & SEM_UNDO) is "true", the value of *sem_op* is subtracted from the calling process's semadj value for the specified semaphore. {ALTER}

If *sem_op* is zero, one of the following will occur: {READ}

If semval is zero, *semop* will return immediately.

If semval is not equal to zero and (*sem_flg* & IPC_NOWAIT) is "true", *semop* will return immediately.

If semval is not equal to zero and (*sem_flg* & IPC_NOWAIT) is "false", *semop* will increment the semzcnt associated with the specified semaphore and suspend execution of the calling process until one of the following occurs:

Semval becomes zero, at which time the value of semzcnt associated with the specified semaphore is decremented.

The semid for which the calling process is awaiting action is removed from the system. When this occurs, *errno* is set equal to EIDRM, and a value of –1 is returned.

The calling process receives a signal that is to be caught. When this occurs, the value of semzcnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in *signal*(2).

Semop will fail if one or more of the following are true for any of the semaphore operations specified by *sops*:

[EINVAL] *Semid* is not a valid semaphore identifier.

[EFBIG] *Sem_num* is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*.

[E2BIG] *Nsops* is greater than the system-imposed maximum.

[EACCES] Operation permission is denied to the calling process (see *intro*(2)).

[EAGAIN] The operation would result in suspension of the calling process but (*sem_flg* & **IPC_NOWAIT**) is "true".

[ENOSPC] The limit on the number of individual processes requesting an **SEM_UNDO** would be exceeded.

[EINVAL] The number of individual semaphores for which the calling process requests a **SEM_UNDO** would exceed the limit.

[ERANGE] An operation would cause a semval to overflow the system-imposed limit.

[ERANGE] An operation would cause a semadj value to overflow the system-imposed limit.

[EFAULT] *Sops* points to an illegal address.

Upon successful completion, the value of sempid for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process.

RETURN VALUE

If *semop* returns due to the receipt of a signal, a value of –1 is returned to the calling process and *errno* is set to EINTR. If it returns due to the removal of a *semid* from the system, a value of –1 is returned and *errno* is set to EIDRM.

Upon successful completion, the value of semval at the time of the call for the last operation in the array pointed to by *sops* is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), semctl(2), semget(2).

NAME

setpgrp – set process group ID

SYNOPSIS

int setpgrp ( )

DESCRIPTION

Setpgrp sets the process group ID of the calling process to the process ID of the calling process and returns the new process group ID.

RETURN VALUE

Setpgrp returns the value of the new process group ID.

SEE ALSO

exec(2), fork(2), getpid(2), intro(2), kill(2), signal(2).

NAME

setuid, setgid – set user and group IDs

SYNOPSIS

int setuid (uid)
int uid;

int setgid (gid)
int gid;

DESCRIPTION

Setuid (*setgid*) is used to set the real user (group) ID and effective user (group) ID of the calling process.

If the effective user ID of the calling process is super-user, the real user (group) ID and effective user (group) ID are set to *uid* (*gid*).

If the effective user ID of the calling process is not super-user, but its real user (group) ID is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

If the effective user ID of the calling process is not super-user, but the saved set-user (group) ID from *exec*(2) is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

Setuid (*setgid*) will fail if the real user (group) ID of the calling process is not equal to *uid* (*gid*) and its effective user ID is not super-user. [EPERM]

The *uid* is out of range. [EINVAL]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

getuid(2), intro(2).

NAME

shmctl – shared memory control operations

SYNOPSIS

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

int shmctl (shmid, cmd, buf)
int shmid, cmd;
struct shmid_ds *buf;
```

DESCRIPTION

Shmctl provides a variety of shared memory control operations as specified by *cmd*. The following *cmd*s are available:

IPC_STAT Place the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure are defined in *intro*(2). {READ}

IPC_SET Set the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure pointed to by *buf*:
shm_perm.uid
shm_perm.gid
shm_perm.mode /* only low 9 bits */

This *cmd* can only be executed by a process that has an effective user ID equal to either that of super-user or to the value of **shm_perm.uid** in the data structure associated with *shmid*.

IPC_RMID Remove the shared memory identifier specified by *shmid* from the system and destroy the shared memory segment and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of super-user or to the value of **shm_perm.uid** in the data structure associated with *shmid*.

Shmctl will fail if one or more of the following are true:

[EINVAL] *Shmid* is not a valid shared memory identifier.

[EINVAL] *Cmd* is not a valid command.

[EACCES] *Cmd* is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process (see *intro*(2)).

[EPERM] *Cmd* is equal to IPC_RMID or IPC_SET and the effective user ID of the calling process is not equal to that of super-user and it is not equal to the value of **shm_perm.uid** in the data structure associated with *shmid*.

[EFAULT] *Buf* points to an illegal address.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

intro(2), shmget(2), shmop(2).

NAME

shmget – get shared memory segment

SYNOPSIS

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

int shmget (key, size, shmflg)
key_t key;
int size, shmflg;
```

DESCRIPTION

Shmget returns the shared memory identifier associated with *key*.

A shared memory identifier and associated data structure and shared memory segment of size *size* bytes (see *intro*(2)) are created for *key* if one of the following are true:

Key is equal to IPC_PRIVATE.

Key does not already have a shared memory identifier associated with it, and (*shmflg* & IPC_CREAT) is "true".

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

Shm_perm.cuid, **shm_perm.uid**, **shm_perm.cgid**, and **shm_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.

The low-order 9 bits of **shm_perm.mode** are set equal to the low-order 9 bits of *shmflg*. **Shm_segsz** is set equal to the value of *size*.

Shm_lpid, **shm_nattch**, **shm_atime**, and **shm_dtime** are set equal to 0.

Shm_ctime is set equal to the current time.

Shmget will fail if one or more of the following are true:

[EINVAL] *Size* is less than the system-imposed minimum or greater than the system-imposed maximum.

[EACCES] A shared memory identifier exists for *key* but operation permission (see *intro*(2)) as specified by the low-order 9 bits of *shmflg* would not be granted.

[EINVAL] A shared memory identifier exists for *key* but the size of the segment associated with it is less than *size* and *size* is not equal to zero.

[ENOENT] A shared memory identifier does not exist for *key* and (*shmflg* & IPC_CREAT) is "false".

[ENOSPC] A shared memory identifier is to be created but the system-imposed limit on the maximum number of allowed shared memory identifiers system wide would be exceeded.

[ENOMEM] A shared memory identifier and associated shared memory segment are to be created but the amount of available physical memory is not sufficient to fill the request.

[EEXIST] A shared memory identifier exists for *key* but ((*shmflg* & IPC_CREAT) and (*shmflg* & IPC_EXCL)) is "true".

RETURN VALUE

Upon successful completion, a non-negative integer, namely a shared memory identifier is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

intro(2), shmctl(2), shmop(2).

NAME

shmop – shared memory operations

SYNOPSIS

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

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr
int shmflg;

int shmdt (shmaddr)
char *shmaddr
```

DESCRIPTION

Shmat attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process. The segment is attached at the address specified by one of the following criteria:

If *shmaddr* is equal to zero, the segment is attached at the first available address as selected by the system.

If *shmaddr* is not equal to zero and (*shmflg* & SHM_RND) is "true", the segment is attached at the address given by (*shmaddr* - (*shmaddr* modulus SHMLBA)).

If *shmaddr* is not equal to zero and (*shmflg* & SHM_RND) is "false", the segment is attached at the address given by *shmaddr*.

The segment is attached for reading if (*shmflg* & SHM_RDONLY) is "true" {READ}, otherwise it is attached for reading and writing {READ/WRITE}.

Shmat will fail and not attach the shared memory segment if one or more of the following are true:

[EINVAL] *Shmid* is not a valid shared memory identifier.

[EACCES] Operation permission is denied to the calling process (see *intro*(2)).

[ENOMEM] The available data space is not large enough to accommodate the shared memory segment.

[EINVAL] *Shmaddr* is not equal to zero, and the value of (*shmaddr* - (*shmaddr* modulus SHMLBA)) is an illegal address.

[EINVAL] *Shmaddr* is not equal to zero, (*shmflg* & SHM_RND) is "false", and the value of *shmaddr* is an illegal address.

[EMFILE] The number of shared memory segments attached to the calling process would exceed the system-imposed limit.

[EINVAL] *Shmdt* detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr*.

[EINVAL] *Shmdt* will fail and not detach the shared memory segment if *shmaddr* is not the data segment start address of a shared memory segment.

RETURN VALUES

Upon successful completion, the return value is as follows:

Shmat returns the data segment start address of the attached shared memory segment.

Shmdt returns a value of 0.

Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), shmctl(2), shmget(2).

NAME

signal – specify what to do upon receipt of a signal

SYNOPSIS

#include <signal.h>

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

DESCRIPTION

Signal allows the calling process to choose one of three ways in which it is possible to handle the receipt of a specific signal. *Sig* specifies the signal and *func* specifies the choice.

Sig can be assigned any one of the following except SIGKILL:

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03*	quit
SIGILL	04*	illegal instruction (not reset when caught)
SIGTRAP	05*	trace trap (not reset when caught)
SIGIOT	06*	IOT instruction
SIGEMT	07*	EMT instruction
SIGFPE	08*	floating point exception
SIGKILL	09	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
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2
SIGCLD	18	death of a child (see *WARNING* below)
SIGPWR	19	power fail (see *WARNING below)*

See below for the significance of the asterisk (*) in the above list.

Func is assigned one of three values: SIG_DFL, SIG_IGN, or a *function address*. The actions prescribed by these values are as follows:

SIG_DFL – terminate process upon receipt of a signal

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(2). In addition a "core image" will be made in the current working directory of the receiving process if *sig* is one for which an asterisk appears in the above list *and* the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask (see *umask*(2))

a file owner ID that is the same as the effective user ID of the receiving process.

a file group ID that is the same as the effective group ID of the receiving process

SIG_IGN – ignore signal

The signal *sig* is to be ignored.

Note: the signal SIGKILL cannot be ignored.

function address – catch signal

Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function pointed to by *func*. The signal number *sig* will be passed as the only argument to the signal-catching function. Before entering the signal-catching function, the value of *func* for the caught signal will be set to SIG_DFL unless the signal is SIGILL, SIGTRAP, or SIGPWR.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted.

When a signal that is to be caught occurs during a *read*, a *write*, an *open*, or an *ioctl* system call on a slow device (like a terminal; but not a file), during a *pause* system call, or during a *wait* system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call may return a –1 to the calling process with *errno* set to EINTR.

Note: The signal SIGKILL cannot be caught.

A call to *signal* cancels a pending signal *sig* except for a pending SIGKILL signal.

Signal will fail if *sig* is an illegal signal number, including SIGKILL. [EINVAL]

RETURN VALUE

Upon successful completion, *signal* returns the previous value of *func* for the specified signal *sig*. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

kill(1), kill(2), pause(2), ptrace(2), wait(2), setjmp(3C).

WARNING

Two other signals that behave differently than the signals described above exist in this release of the system; they are:

SIGCLD	18	death of a child (reset when caught)
SIGPWR	19	power fail (not reset when caught)

There is no guarantee that, in future releases of the operating system or UNIX system, these signals will continue to behave as described below; they are included only for compatibility with some versions of the UNIX system. Their use in new programs is strongly discouraged by Motorola and AT&T.

For these signals, *func* is assigned one of three values: SIG_DFL, SIG_IGN, or a *function address*. The actions prescribed by these values of are as follows:

SIG_DFL - ignore signal

The signal is to be ignored.

SIG_IGN - ignore signal

The signal is to be ignored. Also, if *sig* is SIGCLD, the calling process's child processes will not create zombie processes when they terminate; see *exit*(2).

function address - catch signal

If the signal is SIGPWR, the action to be taken is the same as that described above for *func* equal to *function address*. The same is true if the signal is SIGCLD except, that while the process is executing the signal-catching function, any received SIGCLD signals will be queued and the signal-catching function will be continually reentered until the queue is empty.

The SIGCLD affects two other system calls (*wait*(2), and *exit*(2)) in the following ways:

wait If the *func* value of SIGCLD is set to SIG_IGN and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value of −1 with *errno* set to ECHILD.

exit If in the exiting process's parent process the *func* value of SIGCLD is set to SIG_IGN, the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set SIGCLD to be caught.

BUGS

A user process connot catch a signal caused by an invalid memory reference during a partially completed instruction. Thus SIGSEGV can be ignored or be allowed to terminate the process, but cannot be caught. This bug is due to a temporary implementation problem.

NAME

stat, fstat – get file status

SYNOPSIS

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

int stat (path, buf)
char *path;
struct stat *buf;

int fstat (fildes, buf)
int fildes;
struct stat *buf;
```

DESCRIPTION

Path points to a path name naming a file. Read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable. *Stat* obtains information about the named file. *Stat* works with all files, but does not obtain information peculiar to PILF files (see *syslocal*(2) and *pilf*(5)).

Similarly, *fstat* obtains information about an open file known by the file descriptor *fildes*, obtained from a successful *open*, *creat*, *dup*, *fcntl*, or *pipe* system call.

Buf is a pointer to a *stat* structure into which information is placed concerning the file.

The contents of the structure pointed to by *buf* include the following members:

```
ushort   st_mode;    /* File mode; see mknod(2) */
ino_t    st_ino;     /* Inode number */
dev_t    st_dev;     /* ID of device containing */
                     /* a directory entry for this file */
dev_t    st_rdev;    /* ID of device */
                     /* This entry is defined only for */
                     /* character special or block special files */
short    st_nlink;   /* Number of links */
ushort   st_uid;     /* User ID of the file's owner */
ushort   st_gid;     /* Group ID of the file's group */
off_t    st_size;    /* File size in bytes */
time_t   st_atime;   /* Time of last access */
time_t   st_mtime;   /* Time of last data modification */
time_t   st_ctime;   /* Time of last file status change */
                     /* Times measured in seconds since */
                     /* 00:00:00 GMT, Jan. 1, 1970 */
```

st_atime Time when file data was last accessed. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *read*(2).

st_mtime Time when data was last modified. Changed by the following system calls: *creat*(2), *mknod*(2), *pipe*(2), *utime*(2), and *write*(2).

st_ctime Time when file status was last changed. Changed by the following system calls: *chmod*(2), *chown*(2), *creat*(2), *link*(2), *mknod*(2), *pipe*(2), *unlink*(2), *utime*(2), and *write*(2).

Stat will fail if one or more of the following are true:

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[EFAULT] *Buf* or *path* points to an invalid address.

Fstat will fail if one or more of the following are true:

[EBADF] *Fildes* is not a valid open file descriptor.

[EFAULT] *Buf* points to an invalid address.

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), chown(2), creat(2), link(2), mknod(2), pipe(2), read(2), syslocal(2), time(2), unlink(2), utime(2), write(2).

NAME

stime – set time

SYNOPSIS

int stime (tp)
long *tp;

DESCRIPTION

Stime sets the system's idea of the time and date. *Tp* points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

[EPERM] *Stime* will fail if the effective user ID of the calling process is not super-user.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

time(2).

NAME

swrite – synchronous write on a file

SYNOPSIS

int swrite (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION

Swrite has the same purpose and conventions as *write*(2). The two differ solely in their handling of disk input/output. *Swrite*, unlike *write*, does not give a normal return before physical output is complete. A program that executes an *swrite* can assume that the data is on the disk, not waiting in a buffer pool.

SEE ALSO

creat(2), dup(2), lseek(2), open(2), pipe(2), ulimit(2).

NAME

sync – update super-block

SYNOPSIS

void sync ( )

DESCRIPTION

Sync causes all information in 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 *fsck*, *df*, etc. It is mandatory before a boot.

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

NAME

syslocal – special system requests

SYNOPSIS

#include <syslocal.h>
int syslocal (cmd [, arg] ... **)**
int cmd;

DESCRIPTION

Syslocal executes certain special system calls. The specific call is indicated by the first argument.

System Type

int syslocal(SYSL_SYSTEM);

Return SYSL_MINI for MiniFrame, SYSL_MEGA for System 6600.

Superblock Synchronization

int syslocal(SYSL_RESYNC, devnum)
short devnum

Preserve current contents of superblock. *Devnum* specifies the file system: the high order byte contains the major device number of the character special device; the low order byte contains the minor device number. The action taken differs on System 6300 and System 6600: on System 6300, the system is rebooted; on System 6600, the superblock is reread, replacing the current in-RAM copy of the superblock. Both actions have the effect of preventing the system from writing out the superblock, undoing, for example, the effects of file system repair.

Enable Openi

syslocal(SYSL_OPENI, flag)
int flag

Enables or disables the *openi* system call. *Flag* is 1 for enabling, 0 for disabling. Only the superuser can execute this call, which affects every user on the system.

Application Processor Number (System 6600 Only)

syslocal(SYSL_APNUM)

Return the processor number of the Application Processor on which this process is executing.

Total Application Processors (System 6600 Only)

syslocal(SYSL_TOTAPS)

Return the total number of Application Processors currently running.

Console Control (System 6600 Only)

syslocal(SYSL_CONSOLE, type, action)
int type, action;

Manage Application Processor console. Affects Application Processor on which this process is running. *Type* specifies the type of action, *action* the specific action. Values of *type* are: 0 to query console status, 1 to associate the terminal with a terminal, 2 to control kernel prints, and 3 to control entry to the kernel debugger.

If *type* is 0 and *action* 1, the return value indicates the terminal association of the console: a positive value is the terminal number of the associated terminal; and –1 indicates that no terminal is associated with the console;

If *type* is 0 and *action* is 2, the return value gives the status of kernel diagnostic prints: 0 for off, 1 for on.

If *type* is 0 and *action* is 3, the return value tells whether entry to the kernel debugger is enabled: 0 for no, 1 for yes.

If *type* is 0 and *action* is 4, the contents of the console's circular buffer are written to standard output.

If *type* is 1, *action* indicates a new terminal association for the console. If *action* is 0, terminal assocation is removed. If *action* is -1, the console is associated with the UART ludge port. If *action* is positive, it must be the file descriptor for an open terminal special file; the console is associated with that terminal. If the terminal is under window managment, then the file descriptor refers to one of the windows in that terminal; the console is associated with that particular window. A return value of 0 indicates a successful association, a –1 an unsuccessful association, with the error value in *errno*.

If *type* is 2, *action* controls kernel diagnostic prints: 0 disables, any other value enables.

If *type* is 3, *action* controls access to the kernel debugger: 0 disables, 1 enables, any other value must be a process group whose terminal/window is to have kernel prints enabled. When access to the kernel debugger is enabled, entering a Control-B or Code-B on the console terminal enters the kernel debugger.

Maximum Number of Users

```
syslocal(SYSL_MAXUSERS)
```

Returns maximum number of concurrent logins on the processor on which this process is executing.

PILF File Status (System 6600 Only)

```
#include <types.h>

syslocal(SYSL_PSTAT, name, st_buf)
char *name;
struct p_stat *st_buf;

syslocal(SYSL_PFSTAT, fd, st_buf)
int fd;
struct p_stat *st_buf;

struct p_stat
{
        dev_t   st_dev;
        ino_t   st_ino;
        ushort  st_mode;
        short   st_nlink;
        ushort  st_uid;
        ushort  st_gid;
        dev_t   st_rdev;
        off_t   st_size;
        time_t  st_atime;
        time_t  st_mtime;
        time_t  st_ctime;
        char    st_cluster;
}
```

These calls work exactly like *stat* and *fstat* (see *stat*(2)), except that the status structure has one additional field, *st_cluster*, which gives the cluster size exponent of the file.

Get Process's Cluster Size Exponent (System 6600 Only)

```
syslocal(SYSL_GETCLUS)

syslocal(SYSL_SETCLUS, cluster)
int cluster;
```

A process's cluster size exponent sets the cluster size exponent of any files the process creates on PILF file systems. A process's cluster size exponent can be –1, indicating that the new file's cluster size exponent should be taken from the file system's default cluster size exponent. A new process inherits its parent's exponent.

Syslocal SYSL_GETCLUS returns the process's cluster size exponent.

Syslocal SYSL_SETCLUS sets the process's cluster size exponent to *cluster*.

SEE ALSO

SYSL_CONSOLE
console(1M), console(7).

SYSL_OPENI
openi(2).

SYSL_APNUM
SYSL_TOTAPS
apnum(1M). *System 6600 Administrator's Guide*.

SYSL_RESYNC
fsck(1M).

SYSL_PSTAT
SYSL_PFSTAT
SYSL_GETCLUS
SYSL_SETCLUS
pilf(5).

WARNINGS

Kernel prints and the kernel debugger *syslocal* calls that support them may disappear without notice. Use of kernel prints degrades system performance. Use of the kernel debugger halts normal processing.

NAME

time – get time

SYNOPSIS

long time ((long *) 0)

long time (tloc)
long *tloc;

DESCRIPTION

Time returns the value of time in seconds since 00:00:00 GMT, January 1, 1970.

If *tloc* (taken as an integer) is non-zero, the return value is also stored in the location to which *tloc* points.

[EFAULT] *Time* will fail if *tloc* points to an illegal address.

RETURN VALUE

Upon successful completion, *time* returns the value of time. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

stime(2).

NAME

times – get process and child process times

SYNOPSIS

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

long times (buffer)
struct tms *buffer;

DESCRIPTION

Times fills the structure pointed to by ***buffer*** with time-accounting information. The following are the contents of this structure:

```
struct  tms {
        time_t  tms_utime;
        time_t  tms_stime;
        time_t  tms_cutime;
        time_t  tms_cstime;
};
```

This information comes from the calling process and each of its terminated child processes for which it has executed a *wait*. All times are in 60ths of a second.

Tms_utime is the CPU time used while executing instructions in the user space of the calling process.

Tms_stime is the CPU time used by the system on behalf of the calling process.

Tms_cutime is the sum of the *tms_utimes* and *tms_cutimes* of the child processes.

Tms_cstime is the sum of the *tms_stimes* and *tms_cstimes* of the child processes.

[EFAULT] *Times* will fail if *buffer* points to an illegal address.

RETURN VALUE

Upon successful completion, *times* returns the elapsed real time, in 60ths (100ths) of a second, since an arbitrary point in the past (e.g., system start-up time). This point does not change from one invocation of *times* to another. If *times* fails, a –1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), fork(2), time(2), wait(2).

NAME

ulimit – get and set user limits

SYNOPSIS

long ulimit (cmd, newlimit)
int cmd;
long newlimit;

DESCRIPTION

This function provides for control over process limits. The *cmd* values available are:

1 Get the file size limit of the process. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

2 Set the file size limit of the process to the value of *newlimit*. Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit. *Ulimit* will fail and the limit will be unchanged if a process with an effective user ID other than super-user attempts to increase its file size limit. [EPERM]

3 Get the maximum possible break value. See *brk*(2).

RETURN VALUE

Upon successful completion, a non-negative value is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

brk(2), write(2).

NAME

umask – set and get file creation mask

SYNOPSIS

int umask (cmask)
int cmask;

DESCRIPTION

Umask sets the process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the low-order 9 bits of *cmask* and the file mode creation mask are used.

RETURN VALUE

The previous value of the file mode creation mask is returned.

SEE ALSO

mkdir(1), sh(1), chmod(2), creat(2), mknod(2), open(2).

NAME

umount – unmount a file system

SYNOPSIS

int umount (spec)
char *spec;

DESCRIPTION

Umount requests that a previously mounted file system contained on the block special device identified by *spec* be unmounted. *Spec* is a pointer to a path name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

Umount may be invoked only by the super-user.

Umount will fail if one or more of the following are true:

[EPERM]	The process's effective user ID is not super-user.
[ENXIO]	*Spec* does not exist.
[ENOTBLK]	*Spec* is not a block special device.
[EINVAL]	*Spec* is not mounted.
[EBUSY]	A file on *spec* is busy.
[EFAULT]	*Spec* points to an illegal address.

RETURN VALUE

Upon successful completion a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

mount(2).

NAME

uname – get name of current operating system

SYNOPSIS

#include <sys/utsname.h>

int uname (name)
struct utsname *name;

DESCRIPTION

Uname stores information identifying the current operating system in the structure pointed to by *name.*

Uname uses the structure defined in **<sys/utsname.h>** whose members are:

```
char    sysname[9];
char    nodename[9];
char    release[9];
char    version[9];
char    machine[9];
```

Uname returns a null-terminated character string naming the current operating system in the character array *sysname.* Similarly, *nodename* contains the name that the system is known by on a communications network. *Release* and *version* further identify the operating system. *Machine* contains a standard name that identifies the hardware that the operating system is running on.

[EFAULT] *Uname* will fail if *name* points to an invalid address.

RETURN VALUE

Upon successful completion, a non-negative value is returned. Otherwise, –1 is returned and *errno* is set to indicate the error.

SEE ALSO

uname(1).

NAME

unlink – remove directory entry

SYNOPSIS

int unlink (path)
char *path;

DESCRIPTION

Unlink removes the directory entry named by the path name pointed to be *path*.

The named file is unlinked unless one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write permission is denied on the directory containing the link to be removed.
[EPERM]	The named file is a directory and the effective user ID of the process is not super-user.
[EBUSY]	The entry to be unlinked is the mount point for a mounted file system.
[ETXTBSY]	The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed.
[EROFS]	The directory entry to be unlinked is part of a read-only file system.
[EFAULT]	*Path* points outside the process's allocated address space.

When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file have been closed.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

rm(1), close(2), link(2), open(2).

NAME

ustat – get file system statistics

SYNOPSIS

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

int ustat (dev, buf)
int dev;
struct ustat *buf;
```

DESCRIPTION

Ustat returns information about a mounted file system. *Dev* is a device number identifying a device containing a mounted file system. *Buf* is a pointer to a *ustat* structure that includes to following elements:

```
daddr_t f_tfree;        /* Total free blocks */
ino_t   f_tinode;       /* Number of free inodes */
char    f_fname[6];     /* Filsys name */
char    f_fpack[6];     /* Filsys pack name */
```

Ustat will fail if one or more of the following are true:

[EINVAL] *Dev* is not the device number of a device containing a mounted file system.

[EFAULT] *Buf* points outside the process's allocated address space.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

stat(2), fs(4).

NAME

utime – set file access and modification times

SYNOPSIS

```
#include <sys/types.h>
int utime (path, times)
char *path;
struct utimbuf *times;
```

DESCRIPTION

Path points to a path name naming a file. *Utime* sets the access and modification times of the named file.

If *times* is NULL, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use *utime* in this manner.

If *times* is not NULL, *times* is interpreted as a pointer to a *utimbuf* structure and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the super-user may use *utime* this way.

The times in the following structure are measured in seconds since 00:00:00 GMT, Jan. 1, 1970.

```
struct  utimbuf{
        time_t  actime;     /* access time */
        time_t  modtime;    /* modification time */
};
```

Utime will fail if one or more of the following are true:

[ENOENT]	The named file does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.
[EACCES]	Search permission is denied by a component of the path prefix.
[EPERM]	The effective user ID is not super-user and not the owner of the file and *times* is not NULL.
[EACCES]	The effective user ID is not super-user and not the owner of the file and *times* is NULL and write access is denied.
[EROFS]	The file system containing the file is mounted read-only.
[EFAULT]	*Times* is not NULL and points outside the process's allocated address space.
[EFAULT]	*Path* points outside the process's allocated address space.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

stat(2).

NAME

wait – wait for child process to stop or terminate

SYNOPSIS

int wait (stat_loc)
int *stat_loc;

int wait ((int *)0)

DESCRIPTION

Wait suspends the calling process until one of the immediate children terminates or until a child that is being traced stops because it has hit a break point. The *wait* system call will return prematurely if a signal is received and if a child process stopped or terminated prior to the call on *wait*, return is immediate.

If *stat_loc* (taken as an integer) is non-zero, 16 bits of information called status are stored in the low order 16 bits of the location pointed to by *stat_loc*. *Status* can be used to differentiate between stopped and terminated child processes and if the child process terminated, status identifies the cause of termination and passes useful information to the parent. This is accomplished in the following manner:

If the child process stopped, the high order 8 bits of status will contain the number of the signal that caused the process to stop and the low order 8 bits will be set equal to 0177.

If the child process terminated due to an *exit* call, the low order 8 bits of status will be zero and the high order 8 bits will contain the low order 8 bits of the argument that the child process passed to *exit*; see *exit*(2).

If the child process terminated due to a signal, the high order 8 bits of status will be zero and the low order 8 bits will contain the number of the signal that caused the termination. In addition, if the low order seventh bit (i.e., bit 200) is set, a "core image" will have been produced; see *signal*(2).

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes; see *intro*(2).

Wait will fail and return immediately if one or more of the following are true:

[ECHILD] The calling process has no existing unwaited-for child processes.

[EFAULT] *Stat_loc* points to an illegal address.

RETURN VALUE

If *wait* returns due to the receipt of a signal, a value of –1 is returned to the calling process and *errno* is set to EINTR. If *wait* returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), pause(2), ptrace(2), signal(2).

WARNING

See *WARNING* in *signal*(2).

NAME

write – write on a file

SYNOPSIS

int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

DESCRIPTION

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

Write attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write.

Write will fail and the file pointer will remain unchanged if one or more of the following are true:

[EBADF] *Fildes* is not a valid file descriptor open for writing.

[EPIPE and SIGPIPE signal]
An attempt is made to write to a pipe that is not open for reading by any process.

[EFBIG] An attempt was made to write a file that exceeds the process's file size limit or the maximum file size. See *ulimit*(2).

[EFAULT] *Buf* points outside the process's allocated address space.

[EINTR] A signal was caught during the *write* system call.

[ENOSPC] Additional blocks cannot be allocated to the file becasue the file system has no free blocks or because a PILF file's cluster size exceeds the size of all unallocated clusters.

[EDEADLOCK] A side effect of a previous *locking*(2) call.

If a *write* requests that more bytes be written than there is room for (e.g., the *ulimit* (see *ulimit*(2)) or the physical end of a medium), only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below).

If the file being written is a pipe (or FIFO) and the O_NDELAY flag of the file flag word is set, then write to a full pipe (or FIFO) will return a count of 0. Otherwise (O_NDELAY clear), writes to a full pipe (or FIFO) will block until space becomes available.

RETURN VALUE

Upon successful completion the number of bytes actually written is returned. Otherwise, –1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat(2), dup(2), lseek(2), locking(2), open(2), pipe(2), ulimit(2).