(2)

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 or the NULL pointer; 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*(2) or *ptrace*(2).

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.

(2)

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.

7 E2BIG Arg list too long
An argument list longer than 5,120 bytes is presented to a member of the *exec*(2) 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*(2) [respectively, *write*(2)] 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. Or a system call failed because of insufficient memory or swap space.

12 ENOMEM Not enough space
During an *exec*(2), *brk*(2), or *sbrk*(2), a program asks for more space than the system is able to supply. This may not be a temporary condition; the maximum space size is a system parameter. 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*(2). If this error occurs on a resource associated with Remote File Sharing (RFS), it indicates a memory depletion wich may be temporary, dependent on system activity at the time the call was invoked.

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

14 EFAULT Bad address
The system encountered a hardware fault in attempting to 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*(2).

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*(2).

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*(2) or *kill*(2); reading or writing a file for which *lseek*(2) 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.

(2)

24 EMFILE Too many open files
No process may have more than NOFILES (default 100) descriptors open at a time.

25 ENOTTY Not a character device (or) Not a typewriter
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 or to remove a pure-procedure program that is being executed.

27 EFBIG File too large
The size of a file exceeded the maximum file size or ULIMIT [see *ulimit*(2)].

28 ENOSPC No space left on device
During a *write*(2) to an ordinary file, there is no free space left on the device. In *fcntl*(2), the setting or removing of record locks on a file cannot be accomplished because there are no more record entries left on the system.

29 ESPIPE Illegal seek
An *lseek*(2) 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)].

37-44 Reserved numbers

45 EDEADLK Deadlock
A deadlock situation was detected and avoided. This error pertains to file and record locking.

46 ENOLCK No lock
In *fcntl*(2) the setting or removing of record locks on a file cannot be accomplished because there are no more record entries left on the system.

60 ENOSTR Not a stream
A *putmsg*(2) or *getmsg*(2) system call was attempted on a file descriptor that is not a STREAMS device.

62 ETIME Stream ioctl timeout
The timer set for a STREAMS *ioctl*(2) call has expired. The cause of this error is device specific and could indicate either a hardware or software failure, or perhaps a timeout value that is too short for the specific operation. The status of the *ioctl*(2) operation is indeterminate.

63 ENOSR No stream resources
During a STREAMS *open*(2), either no STREAMS queues or no STREAMS head data structures were available.

64 ENONET Machine is not on the network
This error is Remote File Sharing (RFS) specific. It occurs when users try to advertise, unadvertise, mount, or unmount remote resources while the machine has not done the proper startup to connect to the network.

65 ENOPKG No package
This error occurs when users attempt to use a system call from a package which has not been installed.

(2)

66 EREMOTE Resource is remote
This error is RFS specific. It occurs when users try to advertise a resource which is not on the local machine, or try to mount/unmount a device (or pathname) that is on a remote machine.

67 ENOLINK Virtual circuit is gone
This error is RFS specific. It occurs when the link (virtual circuit) connecting to a remote machine is gone.

68 EADV Advertise error
This error is RFS specific. It occurs when users try to advertise a resource which has been advertised already, or try to stop the RFS while there are resources still advertised, or try to force unmount a resource when it is still advertised.

69 ESRMNT Srmount error
This error is RFS specific. It occurs when users try to stop RFS while there are resources still mounted by remote machines.

70 ECOMM Communication error
This error is RFS specific. It occurs when trying to send messages to remote machines but no virtual circuit can be found.

71 EPROTO Protocol error
Some protocol error occurred. This error is device specific, but is generally not related to a hardware failure.

74 EMULTIHOP Multihop attempted
This error is RFS specific. It occurs when users try to access remote resources which are not directly accessible.

77 EBADMSG Bad message
During a *read*(2), *getmsg*(2), or *ioctl*(2) I_RECVFD system call to a STREAMS device, something has come to the head of the queue that can't be processed. That something depends on the system call:
read(2) - control information or a passed file descriptor.
getmsg(2) - passed file descriptor.
ioctl(2) - control or data information.

83 ELIBACC Cannot access a needed shared library
Trying to *exec*(2) an *a.out* that requires a shared library (to be linked in) and the shared library doesn't exist or the user doesn't have permission to use it.

84 ELIBBAD Accessing a corrupted shared library
Trying to *exec*(2) an *a.out* that requires a shared library (to be linked in) and *exec*(2) could not load the shared library. The shared library is probably corrupted.

85 ELIBSCN .lib section in *a.out* corrupted
Trying to *exec*(2) an *a.out* that requires a shared library (to be linked in) and there was erroneous data in the .lib section of the *a.out*. The .lib section tells *exec*(2) what shared libraries are needed. The *a.out* is probably corrupted.

86 ELIBMAX Attempting to link in more shared libraries than system limit
Trying to *exec*(2) an *a.out* that requires more shared libraries (to be linked in) than is allowed on the current configuration of the system. See the System Administrator's Guide.

87 ELIBEXEC Cannot exec a shared library directly
Trying to *exec*(2) a shared library directly. This is not allowed.

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

(2)

identified by a positive integer (0 to 65535) 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, such as immunity from file permissions, 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 (NOFILES - 1). A process may have no more than NOFILES file descriptors 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, the use of unprintable characters in file names should be avoided.

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.

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 (0070) of the file mode is set.

(2)

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 (0007) 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;
struct   msg *msg_first;
struct   msg *msg_last;
ushort   msg_cbytes;
ushort   msg_qnum;
ushort   msg_qbytes;
ushort   msg_lspid;
ushort   msg_lrpid;
time_t   msg_stime;
time_t   msg_rtime;
time_t   msg_ctime;
```

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 */
ushort   seq;     /* slot usage sequence # */
key_t    key;     /* key */
```

msg *msg_first
is a pointer to the first message on the queue.

msg *msg_last
is a pointer to the last message on the queue.

msg_cbytes
is the current number of bytes on the queue.

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

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
00040	Read by group
00020	Write by group
00004	Read by others
00002	Write by others

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

(2)

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

The effective user ID of the process matches **msg_perm.cuid** or **msg_perm.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 group ID of the process matches **msg_perm.cgid** or **msg_perm.gid** and the appropriate bit of the "group" portion (060) of **msg_perm.mode** is set.

The appropriate bit of the "other" portion (006) 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 */
struct   sem *sem_base;       /* ptr to first semaphore in set */
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   uid;    /* user id */
ushort   gid;    /* group id */
ushort   cuid;   /* creator user id */
ushort   cgid;   /* creator group id */
ushort   mode;   /* r/a permission */
ushort   seq;    /* slot usage sequence number */
key_t    key;    /* key */
```

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.

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 called *sem* 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 which is the actual value of the semphore.

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:

(2)

00400	Read by user
00200	Alter by user
00040	Read by group
00020	Alter by group
00004	Read by others
00002	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.cuid** or **sem_perm.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 group ID of the process matches **sem_perm.cgid** or **sem_perm.gid** and the appropriate bit of the "group" portion (060) of **sem_perm.mode** is set.

The appropriate bit of the "other" portion (006) 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. (Note that these shared memory segments must be explicitly removed by the user after the last reference to them is removed.) 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 */
struct   region *shm_reg;     /*ptr to region structure */
char     pad[4];              /* for swap compatibility */
ushort   shm_lpid;            /* pid of last operation */
ushort   shm_cpid;            /* creator pid */
ushort   shm_nattch;          /* number of current attaches */
ushort   shm_cnattch;         /* used only for shminfo */
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 */
ushort    seq;      /* slot usage sequence # */
key_t     key;      /* key */
```

shm_segsz
specifies the size of the shared memory segment in bytes.

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*(2) operation,

shm_dtime
is the time of the last *shmdt*(2) operation.

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:

(2)

00400	Read by user
00200	Write by user
00040	Read by group
00020	Write by group
00004	Read by others
00002	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.cuid** or **shm_perm.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 group ID of the process matches **shm_perm.cgid** or **shm_perm.gid** and the appropriate bit of the "group" portion (060) of **shm_perm.mode** is set.

The appropriate bit of the "other" portion (06) of **shm_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

STREAMS A set of kernel mechanisms that support the development of network services and data communication *drivers*. It defines interface standards for character input/output within the kernel and between the kernel and user level processes. The STREAMS mechanism is composed of utility routines, kernel facilities and a set of data structures.

Stream A stream is a full-duplex data path within the kernel between a user process and driver routines. The primary components are a *stream head*, a *driver* and zero or more *modules* between the *stream head* and *driver*. A *stream* is analogous to a Shell pipeline except that data flow and processing are bidirectional.

Stream Head In a *stream*, the *stream head* is the end of the *stream* that provides the interface between the *stream* and a user process. The principle functions of the *stream head* are processing STREAMS-related system calls, and passing data and information between a user process and the *stream*.

Driver In a *stream*, the *driver* provides the interface between peripheral hardware and the *stream*. A *driver* can also be a pseudo-*driver*, such as a *multiplexor* or log *driver* [see *log*(7)], which is not associated with a hardware device.

Module A module is an entity containing processing routines for input and output data. It always exists in the middle of a *stream*, between the stream's head and a *driver*. A *module* is the STREAMS counterpart to the commands in a Shell pipeline except that a module contains a pair of functions which allow independent bidirectional (*downstream* and *upstream*) data flow and processing.

Downstream In a *stream*, the direction from *stream head* to *driver*.

Upstream In a *stream*, the direction from *driver* to *stream head*.

Message In a *stream*, one or more blocks of data or information, with associated STREAMS control structures. *Messages* can be of several defined types, which identify the *message* contents. *Messages* are the only means of transferring data and communicating within a *stream*.

Message Queue In a *stream*, a linked list of *messages* awaiting processing by a *module* or *driver*.

Read Queue In a *stream*, the *message queue* in a *module* or *driver* containing *messages* moving *upstream*.

Write Queue In a *stream*, the *message queue* in a *module* or *driver* containing *messages* moving *downstream*.

Multiplexor A multiplexor is a driver that allows *streams* associated with several user processes to be connected to a single *driver*, or several *drivers* to be connected to a single user process. STREAMS does not provide a general multiplexing *driver*, but does provide the facilities for constructing them, and for connecting multiplexed configurations of *streams*.

(2)

SEE ALSO
intro(3).

(2)

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.
[EACCES]	Permission bits of the file mode do not permit the requested access.
[EFAULT]	*Path* points outside the allocated address space for the process.
[EINTR]	A signal was caught during the *access* system call.
[ENOLINK]	*Path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

(2)

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.

SEE ALSO

chmod(2), stat(2).

DIAGNOSTICS

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.

(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 pathname 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]	The file named by *path* is not an ordinary file.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*Path* points to an illegal address.

SEE ALSO

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

DIAGNOSTICS

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

(2)

NAME

advfs – advertise a directory for remote access

SYNOPSIS

int advfs(dir, resource, rwflag)
char *dir;
char *resource;
int rwflag;

DESCRIPTION

advfs advertises *dir*, which points to the pathname of a local directory, under the symbolic name *resource*. *Resource* will be used by remote machines to identify this directory when mounting it on those machines.

The low-order bit of *rwflag* controls write permission by remote machines on the advertised directory. If 1, writing is forbidden, otherwise writing is permitted according to the access permissions on individual files. If the local file system that contains *dir* is mounted read-only, it must be advertised read-only.

advfs may be invoked only by the super-user.

ERRORS

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

[ENONET] The Shared Resource environment has not been started (see *dustart*(1M)).

[EPERM] The effective user ID is not super-user.

[ENOENT] *Dir* does not exist.

[ENOTDIR] A component of *dir* is not a directory.

[EFAULT] *Resource* or *dir* points outside the allocated address space of the process.

[EADV] Directory *dir* is already advertised.

[EBUSY] *Resource* is the name of a currently advertised resource.

[EREMOTE] *Dir* is a remote directory.

[ENOSPC] There are no more entries in the advertise table.

[EROFS] Attempt to advertise a read-only file system as read-write.

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

rmount(2) rumount(2) unadvfs(2)

(2)

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.

SEE ALSO

pause(2), signal(2), sigpause(2), sigset(2).

DIAGNOSTICS

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

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. Newly allocated space is set to zero. If, however, the same memory space is reallocated to the same process its contents are undefined.

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:

[ENOMEM] Such a change would result in more space being allocated than is allowed by the system-imposed maximum process size [see *ulimit*(2)].

[EAGAIN] Total amount of system memory available for a read during physical IO is temporarily insufficient [see *shmop*(2)]. This may occur even though the space requested was less than the system-imposed maximum process size [see *ulimit*(2)].

SEE ALSO

exec(2), shmop(2), ulimit(2), end(3C).

(2)

DIAGNOSTICS

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.

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.
[EINTR]	A signal was caught during the *chdir* system call.
[ENOLINK]	*Path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

chroot(2).

DIAGNOSTICS

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

(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.
020#0	Set group ID on execution if # is **7, 5, 3,** or **1** Enable mandatory file/record locking if # is **6, 4, 2,** or **0**
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 a 410 executable file has the sticky bit (mode bit 01000) set, the operating system will not delete the program text from the swap area when the last user process terminates. If a 413 executable file has the sticky bit set, the operating system will not delete the program text from memory when the last user process terminates. In either case, if the sticky bit is set the text will already be available (either in a swap area or in memory) when the next user of the file executes it, thus making execution faster.

If the mode bit 02000 (set group ID on execution) is set and the mode bit 00010 (execute or search by group) is not set, mandatory file/record

locking will exist on a regular file. This may effect future calls to open(2), creat(2), read(2), and write(2) on this file.

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.
[EINTR]	A signal was caught during the *chmod* system call.
[ENOLINK]	*Path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

chown(2), creat(2), fcntl(2), mknod(2), open(2), read(2), write(2).
chmod(1) in the *User's Reference Manual*.

DIAGNOSTICS

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

(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.
[EINTR]	A signal was caught during the *chown* system call.
[ENOLINK]	*Path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

chmod(2).
chown(1) in the *User's Reference Manual.*

DIAGNOSTICS

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

(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.
[EINTR]	A signal was caught during the *chroot* system call.
[ENOLINK]	*Path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

chdir(2).

DIAGNOSTICS

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

(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*. All outstanding record locks owned by the process (on the file indicated by *fildes*) are removed.

If a STREAMS [see *intro*(2)] file is closed, and the calling process had previously registered to receive a SIGPOLL signal [see *signal*(2) and *sigset*(2)] for events associated with that file [see I_SETSIG in *streamio*(7)], the calling process will be unregistered for events associated with the file. The last *close* for a *stream* causes the *stream* associated with *fildes* to be dismantled. If O_NDELAY is not set and there have been no signals posted for the *stream, close* waits up to 15 seconds, for each module and driver, for any output to drain before dismantling the *stream*. If the O_NDELAY flag is set or if there are any pending signals, *close* does not wait for output to drain, and dismantles the *stream* immediately.

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

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

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

[ENOLINK] *Fildes* is on a remote machine and the link to that machine is no longer ctive.

SEE ALSO

creat(2), dup(2), exec(2), fcntl(2), intro(2), open(2), pipe(2), signal(2), sigset(2).
streamio(7) in the *System Administrator's Reference Manual.*

DIAGNOSTICS

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

(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. 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)].

Upon successful completion, a write-only 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 fails 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]	NOFILES file descriptors are currently open.
[EFAULT]	*Path* points outside the allocated address space of the process.
[ENFILE]	The system file table is full.
[EAGAIN]	The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file [see *chmod*(2)].
[EINTR]	A signal was caught during the *creat* system call.
[ENOLINK]	*Path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[ENOSPC]	The file system is out of inodes.

SEE ALSO

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

DIAGNOSTICS

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.

(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.
[EINTR]	A signal was caught during the *dup* system call.
[EMFILE]	NOFILES file descriptors are currently open.
[ENOLINK]	*Fildes* is on a remote machine and the link to that machine is no longer active.

SEE ALSO

close(2), creat(2), exec(2), fcntl(2), open(2), pipe(2), lockf(3C).

DIAGNOSTICS

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.

(2)

(2)

NAME

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

SYNOPSIS

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

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

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

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

int execlp (file, arg0, arg1, ..., argn, (char *)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, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. 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)].

(2)

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 new process; see *signal*(2).

For signals set by *sigset*(2), *exec* will ensure that the new process has the same system signal action for each signal type whose action is SIG_DFL, SIG_IGN, or SIG_HOLD as the calling process. However, if the action is to catch the signal, then the action will be reset to SIG_DFL, and any pending signal for this type will be held.

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
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)]
file-locks [see *fcntl*(2) and *lockf*(3C)]

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.

(2)

[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 5120 bytes.
[EFAULT]	Required hardware is not present.
[EFAULT]	An a.out that was compiled with the MAU or 32B flag is running on a machine without a MAU or 32B.
[EFAULT]	*Path*, *argv*, or *envp* point to an illegal address.
[EAGAIN]	Not enough memory.
[ELIBACC]	Required shared library does not have execute permission.
[ELIBEXEC]	Trying to *exec*(2) a shared library directly.
[EINTR]	A signal was caught during the *exec* system call.
[ENOLINK]	*Path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

alarm(2), exit(2), fcntl(2), fork(2), nice(2), ptrace(2), semop(2), signal(2), sigset(2), times(2), ulimit(2), umask(2), lockf(3C), a.out(4), environ(5). sh(1) in the *User's Reference Manual*.

DIAGNOSTICS

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.

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 processes' existing child processes and zombie processes is set to 1. This means the initialization process [see *intro*(2)] inherits each of these processes.

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.

If the process has a process, text, or data lock, an *unlock* is performed [see *plock*(2)].

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.

(2)

A death of child signal is sent to the parent.

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

SEE ALSO

acct(2), intro(2), plock(2), semop(2), signal(2), sigset(2), wait(2).

WARNING

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

DIAGNOSTICS

None. There can be no return from an *exit* system call.

(2)

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)].

(2)

F_GETLK Get the first lock which blocks the lock description given by the variable of type *struct flock* pointed to by *arg*. The information retrieved overwrites the information passed to *fcntl* in the *flock* structure. If no lock is found that would prevent this lock from being created, then the structure is passed back unchanged except for the lock type which will be set to F_UNLCK.

F_SETLK Set or clear a file segment lock according to the variable of type *struct flock* pointed to by *arg* [see *fcntl*(5)]. The *cmd* F_SETLK is used to establish read (F_RDLCK) and write (F_WRLCK) locks, as well as remove either type of lock (F_UNLCK). If a read or write lock cannot be set *fcntl* will return immediately with an error value of –1.

F_SETLKW This *cmd* is the same as F_SETLK except that if a read or write lock is blocked by other locks, the process will sleep until the segment is free to be locked.

A read lock prevents any process from write locking the protected area. More than one read lock may exist for a given segment of a file at a given time. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any process from read locking or write locking the protected area. Only one write lock may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

The structure *flock* describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), size (*l_len*), process id (*l_pid*), and RFS system id (*l_sysid*) of the segment of the file to be affected. The process id and system id fields are used only with the F_GETLK *cmd* to return the values for a blocking lock. Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end of file by setting *l_len* to zero (0). If such a lock also has *l_whence* and *l_start* set to zero (0), the whole file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments for either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file

descriptor terminates. Locks are not inherited by a child process in a *fork*(2) system call.

When mandatory file and record locking is active on a file, [see *chmod*(2)], *read* and *write* system calls issued on the file will be affected by the record locks in effect.

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

[EBADF]	*Fildes* is not a valid open file descriptor.
[EINVAL]	*Cmd* is F_DUPFD. *arg* is either negative, or greater than or equal to the configured value for the maximum number of open file descriptors allowed each user.
[EINVAL]	*Cmd* is F_GETLK, F_SETLK, or SETLKW and *arg* or the data it points to is not valid.
[EACCES]	*Cmd* is F_SETLK the type of lock (*l_type*) is a read (F_RDLCK) lock and the segment of a file to be locked is already write locked by another process or the type is a write (F_WRLCK) lock and the segment of a file to be locked is already read or write locked by another process.
[ENOLCK]	*Cmd* is F_SETLK or F_SETLKW, the type of lock is a read or write lock, and there are no more record locks available (too many file segments locked) because the system maximum has been exceeded.
[EDEADLK]	*Cmd* is F_SETLKW, the lock is blocked by some lock from another process, and putting the calling-process to sleep, waiting for that lock to become free, would cause a deadlock.
[EFAULT]	*Cmd* is F_SETLK, *arg* points outside the program address space.
[EINTR]	A signal was caught during the *fcntl* system call.
[ENOLINK]	*Fildes* is on a remote machine and the link to that machine is no longer active.

SEE ALSO

close(2), creat(2), dup(2), exec(2), fork(2), open(2), pipe(2), fcntl(5).

DIAGNOSTICS

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

(2)

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.
F_GETLK	Value other than –1.
F_SETLK	Value other than –1.
F_SETLKW	Value other than –1.

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

WARNINGS

Because in the future the variable *errno* will be set to EAGAIN rather than EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value.

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_IGN, SIG_HOLD,** 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)]
current working directory
root directory
file mode creation mask [see *umask*(2)]
file size limit [see *ulimit*(2)]

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)].

Process locks, text locks and data locks are not inherited by the child [see *plock*(2)].

(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.

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.

[EAGAIN] Total amount of system memory available when reading via raw IO is temporarily insufficient.

SEE ALSO

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

DIAGNOSTICS

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.

NAME

getdents – read directory entries and put in a file system independent format

SYNOPSIS

#include <sys/dirent.h>

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

DESCRIPTION

Fildes is a file descriptor obtained from an *open*(2) or *dup*(2) system call.

getdents attempts to read *nbyte* bytes from the directory associated with *fildes* and to format them as file system independent directory entries in the buffer pointed to by *buf*. Since the file system independent directory entries are of variable length, in most cases the actual number of bytes returned will be strictly less than *nbyte*.

The file system independent directory entry is specified by the *dirent* structure. For a description of this see *dirent*(4).

On devices capable of seeking, *getdents* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *getdents*, the file pointer is incremented to point to the next directory entry.

This system call was developed in order to implement the *readdir*(3X) routine [for a description see *directory*(3X)], and should not be used for other purposes.

getdents 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.

[EINVAL] *nbyte* is not large enough for one directory entry.

[ENOENT] The current file pointer for the directory is not located at a valid entry.

(2)

[ENOLINK] *Fildes* points to a remote machine and the link to that machine is no longer active.

[ENOTDIR] *Fildes* is not a directory.

[EIO] An I/O error occurred while accessing the file system.

SEE ALSO

directory(3X), dirent(4).

DIAGNOSTICS

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. A value of 0 indicates the end of the directory has been reached. If the system call failed, a –1 is returned and *errno* is set to indicate the error.

(2)

NAME

getmsg – get next message off a stream

SYNOPSIS

```
#include <stropts.h>

int getmsg(fd, ctlptr, dataptr, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int *flags;
```

DESCRIPTION

getmsg retrieves the contents of a message [see *intro*(2)] located at the *stream head* read queue from a STREAMS file, and places the contents into user specified buffer(s). The message must contain either a data part, a control part or both. The data and control parts of the message are placed into separate buffers, as described below. The semantics of each part is defined by the STREAMS module that generated the message.

Fd specifies a file descriptor referencing an open *stream*. *Ctlptr* and *dataptr* each point to a *strbuf* structure which contains the following members:

```
int maxlen;     /* maximum buffer length */
int len;        /* length of data        */
char *buf;      /* ptr to buffer         */
```

where *buf* points to a buffer in which the data or control information is to be placed, and *maxlen* indicates the maximum number of bytes this buffer can hold. On return, *len* contains the number of bytes of data or control information actually received, or is 0 if there is a zero-length control or data part, or is -1 if no data or control information is present in the message. *Flags* may be set to the values 0 or RS_HIPRI and is used as described below.

Ctlptr is used to hold the control part from the message and *dataptr* is used to hold the data part from the message. If *ctlptr* (or *dataptr*) is NULL or the *maxlen* field is -1, the control (or data) part of the message is not processed and is left on the *stream head* read queue and *len* is set to -1. If the *maxlen* field is set to 0 and there is a zero-length control (or data) part, that zero-length part is removed from the read queue and *len* is set to 0. If the *maxlen* field is set to 0 and there are more than zero bytes of control (or data) information, that information is left on the read queue and *len* is set to 0. If the *maxlen* field in *ctlptr* or *dataptr* is less than, respectively, the control or data part of the message, *maxlen* bytes are retrieved. In this

(2)

case, the remainder of the message is left on the *stream head* read queue and a non-zero return value is provided, as described below under *DIAGNOSTICS*. If information is retrieved from a *priority* message, *flags* is set to RS_HIPRI on return.

By default, *getmsg* processes the first priority or non-priority message available on the *stream head* read queue. However, a user may choose to retrieve only priority messages by setting *flags* to RS_HIPRI. In this case, *getmsg* will only process the next message if it is a priority message.

If O_NDELAY has not been set, *getmsg* blocks until a message, of the type(s) specified by *flags* (priority or either), is available on the *stream head* read queue. If O_NDELAY has been set and a message of the specified type(s) is not present on the read queue, *getmsg* fails and sets *errno* to EAGAIN.

If a hangup occurs on the *stream* from which messages are to be retrieved, *getmsg* will continue to operate normally, as described above, until the *stream head* read queue is empty. Thereafter, it will return 0 in the *len* fields of *ctlptr* and *dataptr*.

getmsg fails if one or more of the following are true:

[EAGAIN] The O_NDELAY flag is set, and no messages are available.

[EBADF] *Fd* is not a valid file descriptor open for reading.

[EBADMSG] Queued message to be read is not valid for *getmsg*.

[EFAULT] *Ctlptr*, *dataptr*, or *flags* points to a location outside the allocated address space.

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

[EINVAL] An illegal value was specified in *flags*, or the *stream* referenced by *fd* is linked under a multiplexor.

[ENOSTR] A *stream* is not associated with *fd*.

A *getmsg* can also fail if a STREAMS error message had been received at the *stream head* before the call to *getmsg*. The error returned is the value contained in the STREAMS error message.

SEE ALSO

intro(2), read(2), poll(2), putmsg(2), write(2).
STREAMS Primer
STREAMS Programmer's Guide

DIAGNOSTICS

Upon successful completion, a non-negative value is returned. A value of 0 indicates that a full message was read successfully. A return value of MORECTL indicates that more control information is waiting for retrieval. A return value of MOREDATA indicates that more data is waiting for retrieval. A return value of MORECTL|MOREDATA indicates that both types of information remain. Subsequent *getmsg* calls will retrieve the remainder of the message.

(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).

(2)

(2)

NAME

ioctl – control device

SYNOPSIS

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

DESCRIPTION

ioctl performs a variety of control functions on devices and STREAMS. For non-STREAMS files, the functions performed by this call are *device-specific* control functions. The arguments *request* and *arg* are passed to the file designated by *fildes* and are interpreted by the device driver. This control is infrequently used on non-STREAMS devices, with the basic input/output functions performed through the *read*(2) and *write*(2) system calls.

For STREAMS files, specific functions are performed by the *ioctl* call as described in *streamio*(7).

Fildes is an open file descriptor that refers to a device. *Request* selects the control function to be performed and will depend on the device being addressed. *Arg* represents additional information that is needed by this specific device to perform the requested function. The data type of *arg* depends upon the particular control request, but it is either an integer or a pointer to a device-specific data structure.

In addition to device-specific and STREAMS functions, generic functions are provided by more than one device driver, for example, the general terminal interface [see *termio*(7)].

ioctl will fail for any type of file 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 device driver that accepts control functions.

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

ioctl will also fail if the device driver detects an error. In this case, the error is passed through *ioctl* without change to the caller. A particular driver might not have all of the following error cases. Other requests to device drivers will fail if one or more of the following are true.

(2)

[EFAULT] *Request* requires a data transfer to or from a buffer pointed to by *arg*, but some part of the buffer is outside the process's allocated space.

[EINVAL] *Request* or *arg* is not valid for this device.

[EIO] Some physical I/O error has occurred.

[ENXIO] The *request* and *arg* are valid for this device driver, but the service requested can not be performed on this particular subdevice.

[ENOLINK] *Fildes* is on a remote machine and the link to that machine is no longer active.

STREAMS errors are described in *streamio*(7).

SEE ALSO

streamio(7), termio(7) in the *System Administrator's Reference Manual.*

DIAGNOSTICS

Upon successful completion, the value returned depends upon the device control function, but must be a non-negative integer. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

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:

(2)

[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.

SEE ALSO

getpid(2), setpgrp(2), signal(2), sigset(2).
kill(1) in the *User's Reference Manual*.

DIAGNOSTICS

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

(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.

(2)

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

[ENOLINK] *Path* points to a remote machine and the link to that machine is no longer active.

[EMULTIHOP] Components of *path* require hopping to multiple remote machines.

SEE ALSO

unlink(2).

DIAGNOSTICS

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

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. Note that if *fildes* is a remote file descriptor and *offset* is negative, *lseek* will return the file pointer even if it is negative.

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] *Fildes* is not a remote file descriptor, and 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.

SEE ALSO

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

DIAGNOSTICS

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.

(2)

NAME

mkdir – make a directory

SYNOPSIS

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

DESCRIPTION

The routine *mkdir* creates a new directory with the name *path*. The mode of the new directory is initialized from the *mode*. The protection part of the *mode* argument is modified by the process's mode mask [see *umask*(2)].

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to the process's effective group ID. The newly created directory is empty with the possible exception of entries for "." and "..". *mkdir* will fail and no directory will be created 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.
[ENOLINK]	*Path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[EACCES]	Either a component of the path prefix denies search permission or write permission is denied on the parent directory of the directory to be created.
[ENOENT]	The path is longer than the maximum allowed.
[EEXIST]	The named file already exists.
[EROFS]	The path prefix resides on a read-only file system.
[EFAULT]	*Path* points outside the allocated address space of the process.

(2)

[EMLINK] The maximum number of links to the parent directory would be exceeded.

[EIO] An I/O error has occurred while accessing the file system.

DIAGNOSTICS

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

(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
00020#0 set group ID on execution if # is **7**, **5**, **3**, or **1**
enable mandatory file/record locking if # is **6**, **4**, **2**, or **0**
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.

(2)

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.
[ENOSPC]	No space is available.
[EINTR]	A signal was caught during the *mknod* system call.
[ENOLINK]	*Path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

chmod(2), exec(2), umask(2), fs(4).
mkdir(1) in the *User's Reference Manual.*

DIAGNOSTICS

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

WARNING

If **mknod** is used to create a device in a remote directory (Remote File Sharing), the major and minor device numbers are interpreted by the server.

(2)

NAME

mount – mount a file system

SYNOPSIS

#include <sys/mount.h>

int mount (spec, dir, mflag, fstyp)
char *spec, *dir;
int mflag, fstyp;

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. *Fstyp* is the file system type number. The *sysfs*(2) system call can be used to determine the file system type number. Note that if the MS_FSS flag bit of *mflag* is off, the file system type will default to the root file system type. Only if the bit is on will *fstyp* be used to indicate the file system type.

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

The low-order bit of *mflag* 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. It is intended for use only by the *mount*(1M) utility.

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.
[EREMOTE]	*Spec* is remote and cannot be mounted.
[ENOLINK]	*Path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

(2)

[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]	*Spec* is write protected and *mflag* requests write permission.
[ENOSPC]	The file system state in the super-block is not FsOKAY and *mflag* requests write permission.
[EINVAL]	The super block has an invalid magic number or the *fstyp* is invalid or *mflag* is not valid.

SEE ALSO

sysfs(2), umount(2), fs(4).
mount(1M) in the *System Administrator's Reference Manual.*

DIAGNOSTICS

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

(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.cuid** or **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.cuid** or **msg_perm.uid** in the data structure associated with *msqid*.

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

(2)

[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, or to the value of **msg_perm.cuid** or **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.

SEE ALSO

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

DIAGNOSTICS

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

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:

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.

(2)

[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".

SEE ALSO

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

DIAGNOSTICS

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.

(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:

(2)

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.

(2)

[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.

SEE ALSO

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

DIAGNOSTICS

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.

(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 non-negative 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. (The default nice value is 20.) 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 39 and the effective user ID of the calling process is not super-user.

SEE ALSO

exec(2).
nice(1) in the *User's Reference Manual.*

DIAGNOSTICS

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.

(2)

(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*. For non-STREAMS [see *intro*(2)] files, *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.

(2)

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_SYNC When opening a regular file, this flag affects subsequent writes. If set, each *write*(2) will wait for both the file data and file status to be physically updated.

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)].

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.

When opening a STREAMS file, *oflag* may be constructed from O_NDELAY or-ed with either O_RDONLY, O_WRONLY or O_RDWR. Other flag values are not applicable to STREAMS devices and have no effect on them. The value of O_NDELAY affects the operation of STREAMS drivers and certain system calls [see *read*(2), *getmsg*(2), *putmsg*(2) and *write*(2)]. For drivers, the implementation of O_NDELAY is device-specific. Each STREAMS device driver may treat this option differently.

Certain flag values can be set following *open* as described in *fcntl*(2).

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:

[EACCES]	A component of the path prefix denies search permission.
[EACCES]	*oflag* permission is denied for the named file.
[EAGAIN]	The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file [see *chmod* (2)].
[EEXIST]	O_CREAT and O_EXCL are set, and the named file exists.
[EFAULT]	*Path* points outside the allocated address space of the process.
[EINTR]	A signal was caught during the *open* system call.
[EIO]	A hangup or error occurred during a STREAMS *open*.
[EISDIR]	The named file is a directory and *oflag* is write or read/write.
[EMFILE]	NOFILES file descriptors are currently open.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[ENFILE]	The system file table is full.
[ENOENT]	O_CREAT is not set and the named file does not exist.
[ENOLINK]	*Path* points to a remote machine, and the link to that machine is no longer active.
[ENOMEM]	The system is unable to allocate a send descriptor.
[ENOSPC]	O_CREAT and O_EXCL are set, and the file system is out of inodes.
[ENOSR]	Unable to allocate a *stream*.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.
[ENXIO]	O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.

(2)

[ENXIO] A STREAMS module or driver open routine failed.

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

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

SEE ALSO

chmod(2), close(2), creat(2), dup(2), fcntl(2), intro(2), lseek(2), read(2), getmsg(2), putmsg(2), umask(2), write(2).

DIAGNOSTICS

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

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), sigpause(2), wait(2).

(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.

pipe will fail if:

[EMFILE]	NOFILES file descriptors are currently open.
[ENFILE]	The system file table is full.

SEE ALSO

read(2), write(2).
sh(1) in the *User's Reference Manual.*

DIAGNOSTICS

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

(2)

NAME

plock – lock process, text, or data in memory

SYNOPSIS

#include <sys/lock.h>

int plock (op)
int op;

DESCRIPTION

plock allows the calling process to lock its text segment (text lock), its data segment (data lock), or both its text and data segments (process lock) into memory. Locked segments are immune to all routine swapping. *plock* also allows these segments to be unlocked. The effective user ID of the calling process must be super-user to use this call. *Op* specifies the following:

PROCLOCK – lock text and data segments into memory (process lock)

TXTLOCK – lock text segment into memory (text lock)

DATLOCK – lock data segment into memory (data lock)

UNLOCK – remove locks

plock will fail and not perform the requested operation if one or more of the following are true:

[EPERM] The effective user ID of the calling process is not super-user.

[EINVAL] *Op* is equal to **PROCLOCK** and a process lock, a text lock, or a data lock already exists on the calling process.

[EINVAL] *Op* is equal to **TXTLOCK** and a text lock, or a process lock already exists on the calling process.

[EINVAL] *Op* is equal to **DATLOCK** and a data lock, or a process lock already exists on the calling process.

[EINVAL] *Op* is equal to **UNLOCK** and no type of lock exists on the calling process.

(2)

[EAGAIN] Not enough memory.

SEE ALSO

exec(2), exit(2), fork(2).

DIAGNOSTICS

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

(2)

NAME

poll – STREAMS input/output multiplexing

SYNOPSIS

```
#include <stropts.h>
#include <poll.h>

int poll(fds, nfds, timeout)
struct pollfd fds[];
unsigned long nfds;
int timeout;
```

DESCRIPTION

poll provides users with a mechanism for multiplexing input/output over a set of file descriptors that reference open *streams* [see *intro*(2)]. *poll* identifies those *streams* on which a user can send or receive messages, or on which certain events have occurred. A user can receive messages using *read*(2) or *getmsg*(2) and can send messages using *write*(2) and *putmsg*(2). Certain *ioctl*(2) calls, such as I_RECVFD and I_SENDFD [see *streamio*(7)], can also be used to receive and send messages.

Fds specifies the file descriptors to be examined and the events of interest for each file descriptor. It is a pointer to an array with one element for each open file descriptor of interest. The array's elements are *pollfd* structures which contain the following members:

```
int fd;          /* file descriptor */
short events;    /* requested events */
short revents;   /* returned events */
```

where *fd* specifies an open file descriptor and *events* and *revents* are bitmasks constructed by or-ing any combination of the following event flags:

POLLIN A non-priority or file descriptor passing message (see I_RECVFD) is present on the *stream head* read queue. This flag is set even if the message is of zero length. In *revents*, this flag is mutually exclusive with POLLPRI.

POLLPRI A priority message is present on the *stream head* read queue. This flag is set even if the message is of zero length. In *revents*, this flag is mutually exclusive with POLLIN.

(2)

POLLOUT The first downstream write queue in the *stream* is not full. Priority control messages can be sent (see *putmsg*) at any time.

POLLERR An error message has arrived at the *stream head*. This flag is only valid in the *revents* bitmask; it is not used in the *events* field.

POLLHUP A hangup has occurred on the *stream*. This event and POLLOUT are mutually exclusive; a *stream* can never be writable if a hangup has occurred. However, this event and POLLIN or POLLPRI are not mutually exclusive. This flag is only valid in the *revents* bitmask; it is not used in the *events* field.

POLLNVAL The specified *fd* value does not belong to an open *stream*. This flag is only valid in the *revents* field; it is not used in the *events* field.

For each element of the array pointed to by *fds*, *poll* examines the given file descriptor for the event(s) specified in *events*. The number of file descriptors to be examined is specified by *nfds*. If *nfds* exceeds NOFILES, the system limit of open files [see *ulimit*(2)], *poll* will fail.

If the value *fd* is less than zero, *events* is ignored and *revents* is set to 0 in that entry on return from *poll*.

The results of the *poll* query are stored in the *revents* field in the *pollfd* structure. Bits are set in the *revents* bitmask to indicate which of the requested events are true. If none are true, none of the specified bits is set in *revents* when the *poll* call returns. The event flags POLLHUP, POLLERR and POLLNVAL are always set in *revents* if the conditions they indicate are true; this occurs even though these flags were not present in *events*.

If none of the defined events have occurred on any selected file descriptor, *poll* waits at least *timeout* msec for an event to occur on any of the selected file descriptors. On a computer where millisecond timing accuracy is not available, *timeout* is rounded up to the nearest legal value available on that system. If the value *timeout* is 0, *poll* returns immediately. If the value of *timeout* is -1, *poll* blocks until a requested event occurs or until the call is interrupted. *poll* is not affected by the O_NDELAY flag.

poll fails if one or more of the following are true:

[EAGAIN] Allocation of internal data structures failed but request should be attempted again.

[EFAULT] Some argument points outside the allocated address space.

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

[EINVAL] The argument *nfds* is less than zero, or *nfds* is greater than NOFILES.

SEE ALSO

intro(2), read(2), getmsg(2), putmsg(2), write(2).
streamio(7) in the *System Administrator's Reference Manual.*
STREAMS Primer.
STREAMS Programmer's Guide.

DIAGNOSTICS

Upon successful completion, a non-negative value is returned. A positive value indicates the total number of file descriptors that has been selected (i.e., file descriptors for which the *revents* field is non-zero). A value of 0 indicates that the call timed out and no file descriptors have been selected. Upon failure, a value of -1 is returned and *errno* is set to indicate the error.

(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. Then the value of *offset* is subtracted from it, and the remainder multiplied by *scale*. If the resulting number corresponds to an entry inside *buff*, that entry is incremented. An entry is defined as a series of bytes with length *sizeof(short)*.

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 entries in *buff*; 077777 (octal) maps each pair of instruction entries 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.

SEE ALSO

prof(1), times(2), monitor(3C).

DIAGNOSTICS

Not defined.

(2)

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, 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, 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.

(2)

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. 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, 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, 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 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

the condition codes of the Processor Status Word.

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 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.

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

sdb(1), exec(2), signal(2), wait(2).

(2)

NAME

putmsg – send a message on a stream

SYNOPSIS

```
#include <stropts.h>

int putmsg (fd, ctlptr, dataptr, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int flags;
```

DESCRIPTION

putmsg creates a message [see *intro*(2)] from user specified buffer(s) and sends the message to a STREAMS file. The message may contain either a data part, a control part or both. The data and control parts to be sent are distinguished by placement in separate buffers, as described below. The semantics of each part is defined by the STREAMS module that receives the message.

fd specifies a file descriptor referencing an open *stream*. *ctlptr* and *dataptr* each point to a *strbuf* structure which contains the following members:

```
int maxlen;     /* not used */
int len;        /* length of data */
char *buf;      /* ptr to buffer */
```

ctlptr points to the structure describing the control part, if any, to be included in the message. The *buf* field in the *strbuf* structure points to the buffer where the control information resides, and the *len* field indicates the number of bytes to be sent. The *maxlen* field is not used in *putmsg* [see *getmsg*(2)]. In a similar manner, *dataptr* specifies the data, if any, to be included in the message. *flags* may be set to the values 0 or RS_HIPRI and is used as described below.

To send the data part of a message, *dataptr* must be non-NULL and the *len* field of *dataptr* must have a value of 0 or greater. To send the control part of a message, the corresponding values must be set for *ctlptr*. No data (control) part will be sent if either *dataptr* (*ctlptr*) is NULL or the *len* field of *dataptr* (*ctlptr*) is set to -1.

If a control part is specified, and *flags* is set to RS_HIPRI, a *priority* message is sent. If *flags* is set to 0, a non-priority message is sent. If no control part is specified, and *flags* is set to RS_HIPRI, *putmsg* fails and sets *errno* to EINVAL. If no control part and no data part are specified, and *flags* is set

to 0, no message is sent, and 0 is returned.

For non-priority messages, *putmsg* will block if the *stream* write queue is full due to internal flow control conditions. For priority messages, *putmsg* does not block on this condition. For non-priority messages, *putmsg* does not block when the write queue is full and O_NDELAY is set. Instead, it fails and sets *errno* to EAGAIN.

putmsg also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the *stream*, regardless of priority or whether O_NDELAY has been specified. No partial message is sent.

putmsg fails if one or more of the following are true:

[EAGAIN] A non-priority message was specified, the O_NDELAY flag is set and the *stream* write queue is full due to internal flow control conditions.

[EAGAIN] Buffers could not be allocated for the message that was to be created.

[EBADF] *fd* is not a valid file descriptor open for writing.

[EFAULT] *ctlptr* or *dataptr* points outside the allocated address space.

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

[EINVAL] An undefined value was specified in *flags*, or *flags* is set to RS_HIPRI and no control part was supplied.

[EINVAL] The *stream* referenced by *fd* is linked below a multiplexor.

[ENOSTR] A *stream* is not associated with *fd*.

[ENXIO] A hangup condition was generated downstream for the specified *stream*.

[ERANGE] The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost *stream* module. This value is also returned if the control part of the message is larger than the maximum configured size of the control part of a message, or if the data part of a message is larger than the maximum configured size of the data part of a message.

A *putmsg* also fails if a STREAMS error message had been processed by the *stream* head before the call to *putmsg*. The error returned is the value contained in the STREAMS error message.

(2)

SEE ALSO

intro(2), read(2), getmsg(2), poll(2), write(2).
STREAMS Primer.
STREAMS Programmer's Guide.

DIAGNOSTICS

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

(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*(2), *open*(2), *dup*(2), *fcntl*(2), or *pipe*(2) 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.

A *read* from a STREAMS [see *intro*(2)] file can operate in three different modes: "byte-stream" mode, "message-nondiscard" mode, and "message-discard" mode. The default is byte-stream mode. This can be changed using the I_SRDOPT *ioctl* request [see *streamio*(7)], and can be tested with the I_GRDOPT *ioctl*. In byte-stream mode, *read* will retrieve data from the *stream* until it has retrieved *nbyte* bytes, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, *read* retrieves data until it has read *nbyte* bytes, or until it reaches a message boundary. If the *read* does not retrieve all the data in a message, the remaining data are replaced on the *stream*, and can be retrieved by the next *read* or *getmsg*(2) call. Message-discard mode also retrieves data until it has retrieved *nbyte* bytes, or it reaches a message boundary. However, unread data remaining in a message after the *read* returns are discarded, and are not available for a subsequent *read* or *getmsg*.

(2)

When attempting to read from a regular file with mandatory file/record locking set [see *chmod*(2)], and there is a blocking (i.e. owned by another process) write lock on the segment of the file to be read:

If O_NDELAY is set, the read will return a -1 and set errno to EAGAIN.

If O_NDELAY is clear, the read will sleep until the blocking record lock is removed.

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.

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

If O_NDELAY is set, the read will return a -1 and set errno to EAGAIN.

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

When reading from a STREAMS file, handling of zero-byte messages is determined by the current read mode setting. In byte-stream mode, *read* accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. *read* then returns the number of bytes read, and places the zero-byte message back on the *stream* to be retrieved by the next *read* or *getmsg*. In the two other modes, a zero-byte message returns a value of 0 and the message is removed from the *stream*. When a zero-byte message is read as the first message on a *stream*, a value of 0 is returned regardless of the read mode.

A *read* from a STREAMS file can only process data messages. It cannot process any type of protocol message and will fail if a protocol message is encountered at the *stream head*.

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

[EAGAIN] Mandatory file/record locking was set, O_NDELAY was set, and there was a blocking record lock.

[EAGAIN] Total amount of system memory available when reading via raw IO is temporarily insufficient.

[EAGAIN] No message waiting to be read on a *stream* and O_NDELAY flag set.

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

[EBADMSG] Message waiting to be read on a *stream* is not a data message.

[EDEADLK] The read was going to go to sleep and cause a deadlock situation to occur.

[EFAULT] *Buf* points outside the allocated address space.

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

[EINVAL] Attempted to read from a *stream* linked to a multiplexor.

[ENOLCK] The system record lock table was full, so the read could not go to sleep until the blocking record lock was removed.

[ENOLINK] *Fildes* is on a remote machine and the link to that machine is no longer active.

A *read* from a STREAMS file will also fail if an error message is received at the *stream head*. In this case, *errno* is set to the value returned in the error message. If a hangup occurs on the *stream* being read, *read* will continue to operate normally until the *stream head* read queue is empty. Thereafter, it will return 0.

SEE ALSO

creat(2), dup(2), fcntl(2), ioctl(2),intro(2), open(2), pipe(2), getmsg(2). streamio(7), termio(7) in the *System Administrator's Reference Manual.*

DIAGNOSTICS

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.

(2)

NAME

rfstart – start the Remote File Sharing environment

SYNOPSIS

int rfstart(aflag);

DESCRIPTION

rfstart starts the Remote File Sharing software on a machine. It must be called before the *adv*(2), *unadv*(2), *rfstop*(2), *rmount*(2), or *rumount*(2) system calls can be used.

The argument *aflag* determines whether an attempt should be made to authenticate incoming connect requests. If *aflag* equals 0, incoming requests for the Remote File Sharing environment will always be accepted. If *aflag* does not equal 0, incoming requests will be verified before the connect request is accepted.

rfstart(2) may be invoked only by the super-user.

ERRORS

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

[EEXIST] The Remote File Sharing environment has already been started.

[EPERM] The effective user ID is not super-user.

[ECOMM] Communication error.

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

advfs(2) rmount(2) rumount(2) rfstop(2) unadvfs(2)

NAME

rfstop – stop the Remote File Sharing environment

SYNOPSIS

int rfstop()

DESCRIPTION

rfstop stops the Remote File Sharing software on a machine.

rfstop may be invoked only by the super-user.

ERRORS

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

[ENONET] The Remote File Sharing environment is not currently running.

[EPERM] The effective user ID is not super-user.

[EBUSY] This machine still has one or more remote resources mounted locally.

[EADV] This machine still has one or more local resources advertised.

[ESRMNT] One or more of this machine's directories is still mounted by a remote machine.

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

rfstart(2)

(2)

NAME

rmdir – remove a directory

SYNOPSIS

int rmdir (path)
char *path;

DESCRIPTION

rmdir removes the directory named by the path name pointed to by *path*. The directory must not have any entries other than "." and "..".

The named directory is removed unless one or more of the following are true:

[EINVAL]	The current directory may not be removed.
[EINVAL]	The "." entry of a directory may not be removed.
[EEXIST]	The directory contains entries other than those for "." and "..".
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named directory 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 directory to be removed.
[EBUSY]	The directory to be removed is the mount point for a mounted file system.
[EROFS]	The directory entry to be removed is part of a read-only file system.
[EFAULT]	*Path* points outside the process's allocated address space.
[EIO]	An I/O error occurred while accessing the file system.
[ENOLINK]	*Path* points to a remote machine, and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

DIAGNOSTICS

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(2).

rmdir(1), rm(1), and mkdir(1) in the *User's Reference Manual.*

(2)

NAME

rmount – mount a remote directory

SYNOPSIS

```
int rmount(resource, dir, token, rwflag)
char *resource;
char *dir;
char *token;
int  rwflag;
```

DESCRIPTION

rmount mounts a remote directory, identified by *resource*, on the directory identified by *dir*. *Dir* is a pointer to the local pathname, and *resource* is the advertised name of the remote directory.

After successful completion, references to the file *dir* will refer to the remote directory advertised as *resource*.

The low-order bit of *rwflag* controls write permission on the remote directory. If 1, writing is forbidden, otherwise writing is permitted according to the access permissions on individual files.

rmount may be invoked only by the super-user.

ERRORS

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

[ENONET] The Remote File Sharing environment has not been booted.

[EPERM] The effective user ID is not super-user.

[ENOTDIR] A component of the pathname pointed to by *dir* is not a directory.

[EFAULT] *Resource* or *dir* points outside the allocated address space of the process.

[EREMOTE] *Dir* is a remote directory.

[ENXIO] *Dir* is the root of an already mounted local file system.

[ECOMM] Communications error occurred.

[ENOSPC] There are no more mount table entries.

[EBUSY] *Resource* is currently mounted on this machine.

[ENODEV] Can't access remote directory.

[EROFS] Attempt to mount a read-only file system as read-write.

[ENOMEM] Can't allocate space for a new resource.

[ENOENT] Resource not advertised to this machine.

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

advfs(2) rumount(2) unadvfs(2)

(2)

NAME

rumount – unmount a remote directory

SYNOPSIS

```
int rumount(resource)
char *resource;
```

DESCRIPTION

rumount unmounts a remote directory, identified by *resource.*

rumount may be invoked only by the super-user.

ERRORS

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

[ENONET] The Remote File Sharing environment has not been booted.

[EPERM] The effective user ID is not super-user.

[EINVAL] *Resource* is invalid.

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

[ECOMM] Communications error occurred.

[EBUSY] *Resource* is currently mounted on this machine.

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

advfs(2) rmount(2) unadvfs(2)

(2)

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}

(2)

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.cuid** or **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.cuid** or **sem_perm.uid** in the data structure associated with *semid*.

semctl fails 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, or to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid*.
[EFAULT]	*Arg.buf* points to an illegal address.

SEE ALSO

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

DIAGNOSTICS

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.

(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 is 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 fails 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".

SEE ALSO

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

DIAGNOSTICS

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.

(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;
unsigned nsops;
```

DESCRIPTION

semop is used to automatically 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.

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.

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.

(2)

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.

SEE ALSO

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

DIAGNOSTICS

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

(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.

SEE ALSO

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

DIAGNOSTICS

setpgrp returns the value of the new process group ID.

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]

SEE ALSO

getuid(2), intro(2).

DIAGNOSTICS

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

(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 that of super user, or to the value of **shm_perm.cuid** or **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 that of super user, or to the value of **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid*.

SHM_LOCK Lock the shared memory segment specified by *shmid* in memory. This *cmd* can only be executed by a process that has an effective user ID equal to super user.

SHM_UNLOCK
Unlock the shared memory segment specified by *shmid*. This *cmd* can only be executed by a process that has an effective user ID equal to super user.

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, or to the value of **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid*.

[EPERM] *Cmd* is equal to **SHM_LOCK** or **SHM_UNLOCK** and the effective user ID of the calling process is not equal to that of super user.

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

[ENOMEM] *Cmd* is equal to **SHM_LOCK** and there is not enough memory.

SEE ALSO

shmget(2), shmop(2).

DIAGNOSTICS

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

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

(2)

NAME

shmget – get shared memory segment identifier

SYNOPSIS

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

int shmget (key, size, shmflg [,physadr])
key_t key;
int size, shmflg;
int physadr;
```

DESCRIPTION

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

A shared memory identifier and associated data structure and shared memory segment of at least *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.

If (*shmflg* & *IPC_PHYS*) is "true," then *shmget* retrieves the *physadr* argument and creates a shared memory segment starting at that physical memory address. This physical memory must not be within the kernel's free memory pool. When created, a physical shared memory segment does not remove the associated memory from the system free memory pool. Upon removal, the memory is not returned to the system free memory pool.

For physical shared memory, if (*shmflg* & *IPC_NOCLEAR*) is "true," then the shared memory segment is not cleared on the first attach.

For physical shared memory, if (*shmflg* & *IPC_CI*) is "true," then the hardware cache, if any, is inhibited on this shared memory segment.

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 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".

[EPERM] A physical shared memory identifier is to be created but the effective user ID of the calling process is not superuser.

SEE ALSO

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

DIAGNOSTICS

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.

(2)

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

(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*.

Shmdt detaches from the calling process's data segment the shared memory segment located at the address specified 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.

(2)

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

SEE ALSO

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

DIAGNOSTICS

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.

NOTES

The user must explicitly remove shared memory segments after the last reference to them has been removed.

(2)

NAME

signal – specify what to do upon receipt of a signal

SYNOPSIS

```
#include <signal.h>

void (*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[1]	quit
SIGILL	04[1]	illegal instruction (not reset when caught)
SIGTRAP	05[1]	trace trap (not reset when caught)
SIGIOT	06[1]	IOT instruction
SIGEMT	07[1]	EMT instruction
SIGFPE	08[1]	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10[1]	bus error
SIGSEGV	11[1]	segmentation violation
SIGSYS	12[1]	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[2]	death of a child
SIGPWR	19[2]	power fail
SIGPOLL	22[3]	selectable event pending

Func is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. **SIG_DFL**, and **SIG_IGN**, are defined in the include file *signal.h*. Each is a macro that expands to a constant expression of type pointer to function returning *void*, and has a unique value that matches no declarable function.

(2)

The actions prescribed by the values of *func* 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). See NOTE [1] below.

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. Additional arguments are passed to the signal-catching function for hardware-generated signals. 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*(2), a *write*(2), an *open*(2), or an *ioctl*(2) system call on a slow device (like a terminal; but not a file), during a *pause*(2) system call, or during a *wait*(2) 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.

signal will not catch an invalid function argument, *func*, and results are undefined when an attempt is made to execute the function at the bad address.

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]

NOTES

[1] If **SIG_DFL** is assigned for these signals, in addition to the process being terminated, a "core image" will be constructed in the current working directory of the process, if 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

[2] For the signals **SIGCLD** and **SIGPWR**, *func* is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. The actions prescribed by these values are:

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** with one exception: while the process is executing the signal-catching function, any received **SIGCLD** signals will be ignored. (This is the default action.)

In addition, **SIGCLD** affects the *wait*, and *exit* system calls as follows:

(2)

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.

[3] **SIGPOLL** is issued when a file descriptor corresponding to a STREAMS [see *intro*(2)] file has a "selectable" event pending. A process must specifically request that this signal be sent using the I_SETSIG *ioctl* call. Otherwise, the process will never receive **SIGPOLL**.

SEE ALSO

intro(2), kill(2), pause(2), ptrace(2), wait(2), setjmp(3C), sigset(2).
kill(1) in the *User's Reference Manual.*

DIAGNOSTICS

Upon successful completion, *signal* returns the previous value of *func* for the specified signal *sig*. Otherwise, a value of SIG_ERR is returned and *errno* is set to indicate the error. SIG_ERR is defined in the include file *signal.h.*

(2)

NAME

sigset, sighold, sigrelse, sigignore, sigpause – signal management

SYNOPSIS

```
#include <signal.h>

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

int sighold (sig)
int sig;

int sigrelse (sig)
int sig;

int sigignore (sig)
int sig;

int sigpause (sig)
int sig;
```

DESCRIPTION

These functions provide signal management for application processes. *sigset* specifies the system signal action to be taken upon receipt of signal *sig*. This action is either calling a process signal-catching handler *func* or performing a system-defined action.

Sig can be assigned any one of the following values except SIGKILL. Machine or implementation dependent signals are not included (see *NOTES* below). Each value of *sig* is a macro, defined in *<signal.h>*, that expands to an integer constant expression.

SIGHUP	hangup
SIGINT	interrupt
SIGQUIT*	quit
SIGILL*	illegal instruction (not held when caught)
SIGTRAP*	trace trap (not held when caught)
SIGABRT*	abort
SIGFPE*	floating point exception
SIGKILL	kill (can not be caught or ignored)
SIGSYS*	bad argument to system call
SIGPIPE	write on a pipe with no one to read it
SIGALRM	alarm clock
SIGTERM	software termination signal
SIGUSR1	user-defined signal 1

(2)

SIGUSR2	user-defined signal 2
SIGCLD	death of a child (see *WARNING* below)
SIGPWR	power fail (see *WARNING* below)
SIGPOLL	selectable event pending (see *NOTES* below)

See below under SIG_DFL regarding asterisks (*) in the above list.

The following values for the system-defined actions of *func* are also defined in <*signal.h*>. Each is a macro that expands to a constant expression of type pointer to function returning *void* and has a unique value that matches no declarable function.

SIG_DFL – default system action

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

Any pending signal *sig* is discarded and the system signal action is set to ignore future occurrences of this signal type.

SIG_HOLD – hold signal

The signal *sig* is to be held upon receipt. Any pending signal of this type remains held. Only one signal of each type is held.

Otherwise, *func* must be a pointer to a function, the signal-catching handler, that is to be called when signal *sig* occurs. In this case, *sigset* specifies that the process will call this function upon receipt of signal *sig*.

Any pending signal of this type is released. This handler address is retained across calls to the other signal management functions listed here.

When a signal occurs, the signal number *sig* will be passed as the only argument to the signal-catching handler. Before calling the signal-catching handler, the system signal action will be set to SIG_HOLD . During normal return from the signal-catching handler, the system signal action is restored to *func* and any held signal of this type released. If a non-local goto (*longjmp*) is taken, then *sigrelse* must be called to restore the system signal action and release any held signal of this type.

In general, upon return from the signal-catching handler, the receiving process will resume execution at the point it was interrupted. However, when a signal is caught during a *read*(2), a *write*(2), an *open*(2), or an *ioctl*(2) system call during a *sigpause* system call, or during a *wait*(2) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal-catching handler will be executed and then the interrupted system call may return a –1 to the calling process with *errno* set to EINTR.

Sighold and *sigrelse* are used to establish critical regions of code. *Sighold* is analogous to raising the priority level and deferring or holding a signal until the priority is lowered by *sigrelse*. *Sigrelse* restores the system signal action to that specified previously by *sigset*.

Sigignore sets the action for signal *sig* to SIG_IGN (see above).

Sigpause suspends the calling process until it receives a signal, the same as *pause*(2). However, if the signal *sig* had been received and held, it is released and the system signal action taken. This system call is useful for testing variables that are changed on the occurrence of a signal. The correct usage is to use *sighold* to block the signal first, then test the variables. If they have not changed, then call *sigpause* to wait for the signal. *sigset* will fail if one or more of the following are true:

[EINVAL] *Sig* is an illegal signal number (including **SIGKILL**) or the default handling of *sig* cannot be changed.

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

DIAGNOSTICS

Upon successful completion, *sigset* returns the previous value of the system signal action for the specified signal *sig*. Otherwise, a value of SIG_ERR is returned and *errno* is set to indicate the error. SIG_ERR is defined in <*signal.h*>.

(2)

For the other functions, 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(2), pause(2), signal(2), wait(2), setjmp(3C).

WARNING

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

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

For these signals, *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 - 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** with one exception: while the process is executing the signal-catching function, any received **SIGCLD** signals will be ignored. (This is the default action.)

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.

NOTES

SIGPOLL is issued when a file descriptor corresponding to a STREAMS [see *intro*(2)] file has a "selectable" event pending. A process must specifically request that this signal be sent using the I_SETSIG *ioctl*(2) call [see *streamio*(7)]. Otherwise, the process will never receive **SIGPOLL**.

For portability, applications should use only the symbolic names of signals rather than their values and use only the set of signals defined here. The action for the signal **SIGKILL** can not be changed from the default system action.

Specific implementations may have other implementation-defined signals. Also, additional implementation-defined arguments may be passed to the signal-catching handler for hardware-generated signals. For certain hardware-generated signals, it may not be possible to resume execution at the point of interruption.

The signal type **SIGSEGV** is reserved for the condition that occurs on an invalid access to a data object. If an implementation can detect this condition, this signal type should be used.

The other signal management functions, *signal*(2) and *pause*(2), should not be used in conjunction with these routines for a particular signal type.

(2)

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.

Note that in a Remote File Sharing environment, the information returned by *stat* depends upon the user/group mapping set up between the local and remote computers. [See *idload*(1M)].

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_mode The mode of the file as described in the *mknod*(2) system call.

st_ino This field uniquely identifies the file in a given file system. The pair st_ino and st_dev uniquely identifies regular files.

st_dev This field uniquely identifies the file system that contains the file. Its value may be used as input to the *ustat*(2) system call to determine more information about this file system. No other meaning is associated with this value.

st_rdev This field should be used only by administrative commands. It is valid only for block special or character special files and only has meaning on the system where the file was configured.

st_nlink This field should be used only by administrative commands.

st_uid The user ID of the file's owner.

st_gid The group ID of the file's group.

st_size For regular files, this is the address of the end of the file. For pipes or fifos, this is the count of the data currently in the file. For block special or character special, this is not defined.

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.

(2)

[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.

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

[ENOLINK] *Path* points to a remote machine and the link to that machine is no longer active.

[EMULTIHOP] Components of *path* require hopping to multiple remote machines.

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.

[ENOLINK] *Fildes* points to a remote machine and the link to that machine is no longer active.

SEE ALSO

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

DIAGNOSTICS

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

(2)

NAME

statf, fstatf – get file status

SYNOPSIS

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

int statf(path, buf, size)
char *path;
struct statf *buf;
ushort size;

int fstatf(fildes, buf, size)
int fildes;
struct statf *buf;
ushort size;
```

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. *statf* obtains *size* bytes of information about the named file.

Similarly, *fstatf* obtains *size* bytes of information about an open file known by the file descriptor *fildes*, obtained from a successful *open*(2), *creat*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call.

Buf is a pointer to a *statf* 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) */
long      st_perm;        /* Access permissions */
ino_t     st_ino;     /* Inode number */
dev_t     st_dev;     /* ID of a device containing a */
                      /* a directory entry for a file */
dev_t     st_rdev;        /* ID of device */
                      /* This entry is defined only for */
                      /* character 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 */
```

(2)

```
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 */
short     st_ftype;          /* Index into a file system type */
                    /* switch table */
long      st_blksize;        /* Block size */
char      st_node[20];       /* node on which file resides */
long      st_flag;           /* General purpose flag */
```

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

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

st_node — This entry contains the name that the system is known by on a communications network. This name should be unique.

ERRORS

statf 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), time(2), unlink(2), utime(2), write(2), uname(2), open(2), dupf(2), fcntl(2)

(2)

NAME

statfs, fstatfs – get file system information

SYNOPSIS

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

int statfs (path, buf, len, fstyp)
char *path;
struct statfs *buf;
int len, fstyp;

int fstatfs (fildes, buf, len, fstyp)
int fildes;
struct statfs *buf;
int len, fstyp;
```

DESCRIPTION

statfs returns a "generic superblock" describing a file system. It can be used to acquire information about mounted as well as unmounted file systems, and usage is slightly different in the two cases. In all cases, *buf* is a pointer to a structure (described below) which will be filled by the system call, and *len* is the number of bytes of information which the system should return in the structure. *Len* must be no greater than **sizeof (struct statfs)** and ordinarily it will contain exactly that value; if it holds a smaller value the system will fill the structure with that number of bytes. (This allows future versions of the system to grow the structure without invalidating older binary programs.)

If the file system of interest is currently mounted, *path* should name a file which resides on that file system. In this case the file system type is known to the operating system and the *fstyp* argument must be zero. For an unmounted file system *path* must name the block special file containing it and *fstyp* must contain the (non-zero) file system type. In both cases 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.

The *statfs* structure pointed to by *buf* includes the following members:

```
short   f_fstyp;    /* File system type */
short   f_bsize;    /* Block size */
short   f_frsize;   /* Fragment size */
long    f_blocks;   /* Total number of blocks */
long    f_bfree;    /* Count of free blocks */
```

```
long    f_files;    /* Total number of file nodes */
long    f_ffree;    /* Count of free file nodes */
char    f_fname[6]; /* Volume name */
char    f_fpack[6]; /* Pack name */
```

fstatfs is similar, except that the file named by *path* in *statfs* is instead identified by an open file descriptor *filedes* obtained from a successful *open*(2), *creat*(2), *dup*(2), *fcntl*(2), or *pipe*(2) system call.

statfs obsoletes *ustat*(2) and should be used in preference to it in new programs.

statfs and *fstatfs* 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.

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

[EINVAL] *Fstyp* is an invalid file system type; *path* is not a block special file and *fstyp* is nonzero; *len* is negative or is greater than **sizeof (struct statfs)**.

[ENOLINK] *Path* points to a remote machine, and the link to that machine is no longer active.

[EMULTIHOP] Components of *path* require hopping to multiple remote machines.

DIAGNOSTICS

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), time(2), unlink(2), utime(2), write(2), fs(4).

(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.

SEE ALSO

time(2).

DIAGNOSTICS

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

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 re-boot.

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

(2)

NAME

sysfs – get file system type information

SYNOPSIS

```
#include <sys/fstyp.h>
#include <sys/fsid.h>

int sysfs (opcode, fsname)
int opcode;
char *fsname;

int sysfs (opcode, fs_index, buf)
int opcode;
int fs_index;
char *buf;

int sysfs (opcode)
int opcode;
```

DESCRIPTION

sysfs returns information about the file system types configured in the system. The number of arguments accepted by *sysfs* varies and depends on the *opcode*. The currently recognized *opcodes* and their functions are described below:

GETFSIND	translates *fsname*, a null-terminated file-system identifier, into a file-system type index.
GETFSTYP	translates *fs_index*, a file-system type index, into a null-terminated file-system identifier and writes it into the buffer pointed to by *buf*; this buffer must be at least of size **FSTYPSZ** as defined in *<sys/fstyp.h>*.
GETNFSTYP	returns the total number of file system types configured in the system.

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

[EINVAL]	*Fsname* points to an invalid file-system identifier; *fs_index* is zero, or invalid; *opcode* is invalid.

[EFAULT] *Buf* or *fsname* point to an invalid user address.

DIAGNOSTICS

Upon successful completion, *sysfs* returns the file-system type index if the *opcode* is **GETFSIND**, a value of 0 if the *opcode* is **GETFSTYP**, or the number of file system types configured if the *opcode* is **GETNFSTYP**. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

(2)

NAME

sysm68k - machine specific functions

SYNOPSIS

#include <sys/sysm68k.h>

int sysm68k(cmd, arg1, arg2)
int cmd, arg1, arg2;

DESCRIPTION

sysm68k implements machine specific functions. The *cmd* argument determines the function performed. The number of arguments expected is dependent on the function.

Command S68ADDMEM

When *cmd* is S68ADDMEM, the argument is used as the number of pages to add to the free list. Note that this command is available to the superuser only. If more pages are added with this command than were deleted with S68DELMEM, only the amount previously deleted will be added back.

Command S68BCACHEOFF

When *cmd* is S68BCACHEOFF, the cache on the **MVME131** board is disabled. Note this command is available to the superuser only.

Please note that board caching is only available on the MVME131, MVME132, MVME132xt and MVME141. This command will fail if attempted when executing on other processor boards.

Command S68BCACHEON

When *cmd* is S68BCACHEON, argument is used as the value to be written to the cache mask register on the **MVME131** and the cache is then enabled. Note this command is available to the superuser only.

Please note that board caching is only available on the MVME131, MVME132, MVME132xt and MVME141. This command will fail if attempted when executing on other processor boards.

Command S68BRDSTAT

When *cmd* is S68BRDSTAT, a structure containing processor board specific status is returned. Two arguments are required. The first argument is a pointer to a *boardid* structure into which information is placed concerning the processor board. Refer to */usr/include/sys/mvmecpu.h* for a declaration of this structure. The second argument is the number of bytes to be transferred. This should be the size of the structure for the first argument or smaller. This command may be executed by any user. This command is only available on some of the newer processor boards. Refer to the specific BUG manual for the processor board being used and to the ROM calls available, which may return similar detailed information about the processor board.

Command S68CACHERD

When *cmd* is executed the current mask used for setting the cacr (cache control register) is returned. No other arguments are necessary. This command may be executed by any user. Refer to the next command, S68CACHESET, and to the *Motorola MC68020 or MC68030 User's Manual* for a description of the control bit values returned.

Command S68CACHESET

When *cmd* is S68CACHESET, the processor instruction and data caches may be enabled or disabled with various options. A value for setting the cacr (cache control register) is passed as the only argument. If the command is successful, this value is returned as the return value. This command may only be executed as superuser. The following control bits are available in the cacr with the MC68030 microprocessor:

CACR_EI	0x0001	Enable Instruction Cache
CACR_FI	0x0002	Freeze Instruction Cache
CACR_CEI	0x0004	Clear Entry in Instruction Cache
CACR_CI	0x0008	Clear Instruction Cache
CACR_IBE	0x0010	Instruction Burst Enable
CACR_ED	0x0100	Enable Data Cache
CACR_FD	0x0200	Freeze Data Cache
CACR_CED	0x0400	Clear Entry in Data Cache
CACR_CD	0x0800	Clear Data Cache
CACR_DBE	0x1000	Data Burst Enable

(2)

CACR_WA 0x2000 Write Allocate

Refer to */usr/include/sys/sysm68k.h* for a declaration of these defines and other cacr related information. Any flags relating to the data cache are only available on the MC68030 and not on the MC68020. The meaning and use of these flags is described in the *Motorola MC68020 or MC68030 User's Manual*. However, when running under **sysV68** only certain limited combinations of the above will be legal. Only the following flags will be allowed to be turned on (or set to a 1):

CACR_EI	Enable Instruction Cache
CACR_CI	Clear Instruction Cache
CACR_IBE	Instruction Burst Enable
CACR_ED	Enable Data Cache
CACR_CD	Clear Data Cache
CACR_DBE	Data Burst Enable
CACR_WA	Write Allocate

If executing on an MC68020 only those flag bits relating to the instruction cache may be set; all others must be set to zero. Any unused bits within the control register are reserved for future use and must be set to zero, as well. Because of the interrelationships of these control flags the following rules define the legal combinations:

1. If CACR_EI is set then CACR_CI must be set as well.
2. If CACR_IBE is set then CACR_EI must be set also.
3. If CACR_ED is set then CACR_CD and CACR_WA must also be set.
4. If CACR_DBE is set then CACR_ED must be set also.

Command S68CONT

When *cmd* is S68CONT, the kernel will continue with the instruction that was interrupted by a bus error signal to the calling routine.

Command S68CPUBRD

When *cmd* is S68CPUBRD, no arguments are expected. A value corresponding to the processor board on which the operating system is running is returned. Refer to **/usr/include/sys/mvmecpu.h** for the mnemonic names used for the cpu board values.

Command S68DELMEM

When *cmd* is S68DELMEM, the argument is used as the number of pages to delete from the free list. Note that this command is available to the superuser only. This command is intended to allow stress tests to verify system behaviour with low free memory.

Command S68FPEX

When *cmd* is S68FPEX, the floating point operand that caused the floating point exception is returned to the user at the address specified by arg1. This command should be executed only after a floating point exception has been indicated to the caller, otherwise an undetermined operand will be returned to the user.

Command S68FPHW

When *cmd* is S68FPHW, a flag is set at the address specified by the argument that indicates whether or not the floating point hardware chip is present on the system. A flag of NOFPHW will be stored if there isn't a floating point chip, a flag of MC68881 will be stored if there is.

Command S68ICACHEOFF

When *cmd* is S68ICACHEOFF, the internal cache of the MC68020 chip is disabled. Note this command is available to the superuser only.

Command S68ICACHEON

When *cmd* is S68ICACHEON, the internal cache of the MC68020 chip is enabled. Note this command is available to the superuser only.

Command S68MEMSIZE

When *cmd* is S68MEMSIZE, no arguments are expected. The size of the virtual memory space and the amount of physical memory (in bytes) are returned.

(2)

Command S68RTODC

When *cmd* is S68RTODC, the value of the real time clock (rtc) is returned to the address specified by the argument. If there is no real time clock on the system, the current time is returned. Note that this command is available to the superuser only.

Command S68SETNAME

When *cmd* is S68SETNAME, the argument is expected to be a pointer to a character string. The system name and node name are set to the character string specified by the argument. Note that this command is available to the superuser only.

Command S68STACK

>>> This system call is obsolete and is included only for compatibility with previous releases.

When *cmd* is S68STACK, the available stack space is increased by the number of bytes (rounded to the nearest page boundary). If this system call succeeds, the new value of the stack pointer is returned.

Command S68STIME

When *cmd* is S68STIME, the argument is used as the new value for the system time and date. The argument contains the time as measured in seconds from **00:00:00 GMT January 1,1970**. Note that this command is only available to the superuser. This command is redundant in that stime(2) may also be used to set the system time but this command is included for compatibility with previous releases.

Command S68SWAP

When *cmd* is S68SWAP, individual swapping areas may be added or deleted, or the current areas determined. The address of an appropriately primed swap buffer is passed as the only argument. (Refer to *sys/swap.h* header file for details of loading the buffer.)

The format of the swap buffer is:

```
struct swapint {
char            si_cmd;     /* command: list, add, delete*/
char            *si_buf;    /* swap file path pointer*/
int             sw_swplo;   /* start block*/
int             si_nblks;   /* swap size*/
}
```

Note that the add and delete options of the command may only be exercised by the superuser.

Typically, a swap area is added by a single call to **sysm68k**. First, the swap buffer is primed with appropriate entries for the structure members. Then **sysm68k** is invoked.

```
#include <sys/sysm68k.h> #include <sys/swap.hZ

struct swapint swapbuf;

sysm68k(S68SWAP, &swapbuf);
```

Command S68TODCSTAT

When *cmd* is S68TODCSTAT, no arguments are expected. The integer return value reflects the status of the time of day clock. A useful time of day clock is indicated by a return value equal to GOOD_TODC.

Command S68WPOSTOFF

When *cmd* is S68WPOSTOFF, write-posting is disabled. No other arguments are necessary. This command may only be executed by superuser. This command is only available on the MVME141 processor board, however, it may well become available in future processor boards. Refer to the *MVME141 User's Manual* for an in depth description of write-posting.

Command S68WPOSTON

When *cmd* is S68WPOSTON, write-posting is enabled. No other arguments are necessary. This command may only be executed by superuser. This command is only available on the MVME141 processor board, however, it may well become available in future processor boards. Refer to

(2)

the *MVME141 User's Manual* for an in depth description of write-posting.

SEE ALSO

swap(1M) in the *System Administrator's Reference Manual.*

NAME

time – get time

SYNOPSIS

#include <sys/types.h>

time_t time (tloc)
long *tloc;

DESCRIPTION

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

If *tloc* is non-zero, the return value is also stored in the location to which *tloc* points.

SEE ALSO

stime(2).

WARNING

time fails and its actions are undefined if *tloc* points to an illegal address.

DIAGNOSTICS

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

(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 reported in clock ticks per second. Clock ticks are a system-dependent parameter. The specific value for an implementation is defined by the variable HZ, found in the include file param.h.

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.

SEE ALSO

exec(2), fork(2), time(2), wait(2).

DIAGNOSTICS

Upon successful completion, *times* returns the elapsed real time, in clock ticks per second, from 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. On a VME Delta Series Computer clock ticks occur 100 times per second.

(2)

(2)

NAME

uadmin – administrative control

SYNOPSIS

#include <sys/uadmin.h>

int uadmin (cmd, fcn, mdep)
int cmd, fcn, mdep;

DESCRIPTION

uadmin provides control for basic administrative functions. This system call is tightly coupled to the system administrative procedures and is not intended for general use. The argument *mdep* is provided for machine-dependent use and is not defined here.

As specified by *cmd*, the following commands are available:

A_SHUTDOWN The system is shutdown. All user processes are killed, the buffer cache is flushed, and the root file system is unmounted. The action to be taken after the system has been shut down is specified by *fcn*. The functions are generic; the hardware capabilities vary on specific machines.

- AD_HALT Halt the processor and turn off the power.
- AD_BOOT Reboot the system, using /sysV68.
- AD_IBOOT Interactive reboot; user is prompted for system name.

A_REBOOT The system stops immediately without any further processing. The action to be taken next is specified by *fcn* as above.

A_REMOUNT The root file system is mounted again after having been fixed. This should be used only during the startup process.

uadmin fails if any of the following are true:

[EPERM] The effective user ID is not super-user.

(2)

DIAGNOSTICS

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

A_SHUTDOWN	Never returns.
A_REBOOT	Never returns.
A_REMOUNT	0

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

(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 regular 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 regular 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* fails and the limit is unchanged if a process with an effective user ID other than super-user attempts to increase its regular file size limit. [EPERM]

3 Get the maximum possible break value [see *brk*(2)].

4 Get the maximum possible number of file descriptors that may be used by a process at a time. This is normally configured to NOFILES. (See *system*(1M) for reconfiguring the value of NOFILES.)

SEE ALSO

brk(2), close(2), creat(2), dup(2), open(2), sysgen(1M), write(2).

WARNING

ulimit is effective in limiting the growth of regular files. Pipes are currently limited to 5,120 bytes.

DIAGNOSTICS

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

(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.

SEE ALSO

chmod(2), creat(2), mknod(2), open(2).
mkdir(1), sh(1) in the *User's Reference Manual.*

DIAGNOSTICS

The previous value of the file mode creation mask is returned.

(2)

NAME

umount – unmount a file system

SYNOPSIS

int umount (file)
char *file;

DESCRIPTION

umount requests that a previously mounted file system contained on the block special device or directory identified by *file* be unmounted. *File* 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.
[EINVAL]	*File* does not exist.
[EINVAL]	*File* is not a block special device.
[EINVAL]	*File* is not mounted.
[EBUSY]	A file on *file* is busy.
[EFAULT]	*File* points to an illegal address.
[EREMOTE]	*File* is remote.
[ENOLINK]	*File* is on a remote machine, and the link to that machine is no longer active.
[EMULTIHOP]	Components of the path pointed to by *file* require hopping to multiple remote machines.

SEE ALSO

mount(2).

DIAGNOSTICS

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

(2)

NAME

unadv – unadvertise a directory

SYNOPSIS

int unadv(resource)
char *resource;

DESCRIPTION

unadv unadvertises *resource,* which is the advertised domain name of a local directory. *unadv* withdraws the directory so that future attempts to remotely mount it will fail. It does not affect remote users who already have *resource* mounted; they may continue to access the directory normally.

unadv may be invoked only by the super-user.

ERRORS

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

[ENONET] The Shared Resource environment has not been started.

[EPERM] The effective user ID is not super-user.

[ENODEV] *Resource* is not advertised.

[EFAULT] *Resource* 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

advfs(2), rmount(2)

NAME

uname – get name of current SYSTEM V/68 system

SYNOPSIS

#include <sys/utsname.h>

int uname (name)
struct utsname *name;

DESCRIPTION

uname stores information identifying the current SYSTEM V/68 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 SYSTEM V/68 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 SYSTEM V/68 system is running on.

[EFAULT] *uname* will fail if *name* points to an invalid address.

SEE ALSO

uname(1) in the *User's Reference Manual.*

DIAGNOSTICS

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

(2)

NAME

unlink – remove directory entry

SYNOPSIS

int unlink (path)
char *path;

DESCRIPTION

unlink removes the directory entry named by the path name pointed to by *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.
[EINTR]	A signal was caught during the *unlink* system call.
[ENOLINK]	*Path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

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.

SEE ALSO

close(2), link(2), open(2).
rm(1) in the *User's Reference Manual.*

DIAGNOSTICS

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

(2)

NAME

ustat – get file system statistics

SYNOPSIS

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

int ustat (dev, buf)
dev_t 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 the 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.

[EINTR] A signal was caught during a *ustat* system call.

[ENOLINK] *Dev* is on a remote machine and the link to that machine is no longer active.

[ECOMM] *Dev* is on a remote machine and the link to that machine is no longer active.

SEE ALSO

stat(2), fs(4).

DIAGNOSTICS

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

(2)

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.

(2)

[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.
[EINTR]	A signal was caught during the *utime* system call.
[ENOLINK]	*Path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

SEE ALSO

stat(2).

DIAGNOSTICS

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

NAME

wait – wait for child process to stop or terminate

SYNOPSIS

int wait (stat_loc)
int *stat_loc;

DESCRIPTION

wait suspends the calling process until 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* 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)].

(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.

SEE ALSO

exec(2), exit(2), fork(2), intro(2), pause(2), ptrace(2), signal(2).

WARNING

wait fails and its actions are undefined if *stat_loc* points to an invalid address.

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

DIAGNOSTICS

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.

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*(2), *open*(2), *dup*(2), *fcntl*(2), or *pipe*(2) 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.

For regular files, if the O_SYNC flag of the file status flags is set, the write will not return until both the file data and file status have been physically updated. This function is for special applications that require extra reliability at the cost of performance. For block special files, if O_SYNC is set, the write will not return until the data has been physically updated.

A write to a regular file will be blocked if mandatory file/record locking is set [see *chmod*(2)], and there is a record lock owned by another process on the segment of the file to be written. If O_NDELAY is not set, the write will sleep until the blocking record lock is removed.

For STREAMS [see *intro*(2)] files, the operation of *write* is determined by the values of the minimum and maximum *nbyte* range ("packet size") accepted by the *stream*. These values are contained in the topmost *stream* module. Unless the user pushes [see I_PUSH in *streamio*(7)] the topmost module, these values can not be set or tested from user level. If *nbyte* falls within the packet size range, *nbyte* bytes will be written. If *nbyte* does not fall within the range and the minimum packet size value is zero, *write* will

(2)

break the buffer into maximum packet size segments prior to sending the data downstream (the last segment may contain less than the maximum packet size). If *nbyte* does not fall within the range and the minimum value is non-zero, *write* will fail with *errno* set to ERANGE. Writing a zero-length buffer (*nbyte* is zero) sends zero bytes with zero returned.

For STREAMS files, if O_NDELAY is not set and the *stream* can not accept data (the *stream* write queue is full due to internal flow control conditions), *write* will block until data can be accepted. O_NDELAY will prevent a process from blocking due to flow control conditions. If O_NDELAY is set and the *stream* can not accept data, *write* will fail. If O_NDELAY is set and part of the buffer has been written when a condition in which the *stream* can not accept additional data occurs, *write* will terminate and return the number of bytes written.

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

[EAGAIN] Mandatory file/record locking was set, O_NDELAY was set, and there was a blocking record lock.

[EAGAIN] Total amount of system memory available when reading via raw IO is temporarily insufficient.

[EAGAIN] Attempt to write to a *stream* that can not accept data with the O_NDELAY flag set.

[EBADF] *fildes* is not a valid file descriptor open for writing.

[EDEADLK] The write was going to go to sleep and cause a deadlock situation to occur.

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

[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)].

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

[EINVAL] Attempt to write to a *stream* linked below a multiplexor.

[ENOLCK] The system record lock table was full, so the write could not go to sleep until the blocking record lock was removed.

[ENOLINK] *fildes* is on a remote machine and the link to that machine is no longer active.

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

[ENXIO] A hangup occurred on the *stream* being written to.

[EPIPE and SIGPIPE signal]
An attempt is made to write to a pipe that is not open for reading by any process.

[ERANGE] Attempt to write to a *stream* with *nbyte* outside specified minimum and maximum write range, and the minimum value is non-zero.

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.

A write to a STREAMS file can fail if an error message has been received at the stream head. In this case, *errno* is set to the value included in the error message.

SEE ALSO

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

DIAGNOSTICS

Upon successful completion the number of bytes actually written is returned. Otherwise, –1 is returned and *errno* is set to indicate the error.

(2)