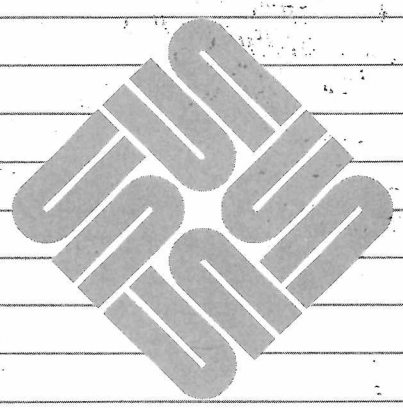
sun microsystems®

SunOS Reference Manual



Part Number: 800-3827-10
Revision A of 27 March, 1990

NAME

intro – introduction to system services and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section describes all of the system calls.

A 2V section number means one or more of the following:

- The man page documents System V behavior only.
- The man page documents default SunOS behavior and System V behavior as it differs from the default behavior. These System V differences are presented under SYSTEM V section headers.
- The man page documents behavior compliant with *IEEE Std 1003.1-1988* (POSIX.1).

Compile programs for the System V environment using **/usr/5bin/cc**. Compile programs for the default SunOS environment using **/usr/bin/cc**. The following man pages describe the various environments provided by Sun: **lint**(1V), **ansic**(7V), **bsd**(7), **posix**(7V), **sunos**(7V), **svidii**(7V), **svidiii**(7V), **xopen**(7V).

Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible return value. This is almost always '–1'; the individual descriptions specify the details. An error code 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. Note: several system calls overload the meanings of these error numbers, and the meanings must be interpreted according to the type and circumstances of the call. See **ERROR CODES** below for a list of system error codes.

As with normal arguments, all return codes and values from functions are of type integer unless otherwise noted.

The rest of this man page is organized as follows:

SYSTEM PARAMETERS	System limits, values and options.
DEFINITIONS	System abstractions and services.
STREAMS	Modular communication between software layers (tty system, networking).
SYSTEM V IPC	System V shared memory, semaphores, and messages.
ERROR CODES	A list of system error codes with descriptions.
LIST OF SYSTEM CALLS	A list of all system calls with brief descriptions.

SYSTEM PARAMETERS

Sections 2 and 3 support a naming convention for those system parameters that may change from one object to another (for example, path name length may is 255 on a UFS file system but may be 14 on an NFS file system exported by a System V based server). Typically, the system has to be queried (using **pathconf**(2V), **fpathconf**(), or **sysconf**(2V)) to retrieve the parameter of interest. The parameters have conceptual names such as PATH_MAX. These names are defined in header files if and only if they are invariant across all file systems and releases of the operating system, that is, very rarely. Because they *may* be defined and/or available from the system calls, there have to be separate names for the parameters and their values. The notation {PATH_MAX} denotes the value of the parameter PATH_MAX. Do not confuse this with **_PC_PATH_MAX**, the name that is passed to the system call to retrieve the value:

```
maxpathlen = pathconf(".", _PC_PATH_MAX);
```

See **pathconf**(2V), and **sysconf**(2V) for further information about these parameters.

DEFINITIONS

Controlling Terminal

A terminal that is associated with a session. Each session may have at most one controlling terminal; a terminal may be the controlling terminal of at most one session. The controlling terminal is used to direct signals (such as interrupts and job control signals) to the appropriate processes by way of the tty's process group. Controlling terminals are assigned when a session leader opens a terminal file that is not currently a controlling terminal.

Descriptor

An integer assigned by the system when a file is referenced by **open**(2V), **dup**(2V), or **pipe**(2V) or a socket is referenced by **socket**(2) or **socketpair**(2) that uniquely identifies an access path to that file or socket from a given process or any of its children.

Directory

A directory is a special type of file that contains entries that are references to other files. 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.

Effective User ID, Effective Group ID, and Access Groups

Access to system resources is governed by three values: the effective user ID, the effective group ID, and the supplementary group ID.

The effective user ID and effective group ID are initially the process's real user ID and real group ID respectively. Either may be modified through execution of a set-user-ID or set-group-ID file (possibly by one of its ancestors) (see **execve**(2V)).

The supplementary group ID are an additional set of group ID's used only in determining resource accessibility. Access checks are performed as described below in **File Access Permissions**.

File Access Permissions

Every file in the file system has a set of access permissions. These permissions are used in determining whether a process may perform a requested operation on the file (such as opening a file for writing). Access permissions are established at the time a file is created. They may be changed at some later time through the **chmod**(2V) call.

File access is broken down according to whether a file may be: read, written, or executed. Directory files use the execute permission to control if the directory may be searched.

File access permissions are interpreted by the system as they apply to three different classes of users: the owner of the file, those users in the file's group, anyone else. Every file has an independent set of access permissions for each of these classes. When an access check is made, the system decides if permission should be granted by checking the access information applicable to the caller.

Read, write, and execute/search permissions on a file are granted to a process if:

The process's effective user ID is that of the super-user.

The process's effective user ID matches the user ID of the owner of the file and the owner permissions allow the access.

The process's effective user ID does not match the user ID of the owner of the file, and either the process's effective group ID matches the group ID of the file, or the group ID of the file is in the process's supplementary group IDs, and the group permissions allow the access.

Neither the effective user ID nor effective group ID and supplementary group IDs of the process match the corresponding user ID and group ID of the file, but the permissions for "other users" allow access.

Otherwise, permission is denied.

File Name

Names consisting of up to {NAME_MAX} characters may be used to name an ordinary file, special file, or directory.

These characters may be selected from the set of all ASCII character excluding \0 (null) and the ASCII code for / (slash). (The parity bit, bit 8, must be 0.)

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

Parent Process ID

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

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. The total length of a path name must be less than **{PATH_MAX}** characters.

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

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

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

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

A slash, by itself, names the root directory. A dot (.) names the current working directory.

A null path name also refers to the current directory. However, this is not true of all UNIX systems. (On such systems, accidental use of a null path name in routines that do not check for it may corrupt the current working directory.) For portable code, specify the current directory explicitly using '"."', rather than '""'.

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 the description of **killpg**() on **kill**(2V)) and the job control mechanisms of **csh**(1). Process groups exist from their creation until the last member is reaped (that is, a parent issued a call to **wait**(2V)).

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 0 to MAXPID (see **<sys/param.h>**).

Real User ID and Real Group ID

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

Each user is also a member of one or more groups. One of these groups is distinguished from others and used in implementing accounting facilities. The positive integer corresponding to this distinguished group is termed the real group ID.

All processes have a real user ID and real group ID. These are initialized from the equivalent attributes of the process that created it.

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 is used as the starting point for absolute path name resolution. The current working directory is used as the starting point for relative path name resolution. A process's root directory need not be (but typically is) the root directory of the root file system.

Session

Each process is a member of a session. A session is associated with each controlling terminal in the system, such as login shells and windows. Each process is created in the session of its parent. A process may alter its session using **setsid**(2V) if it is not already a session leader. The system supports session IDs. A session leader is a process having process ID equal to process group ID equal to session ID. Only a session leader may acquire a controlling terminal. In SunOS Release 4.1, processes are created in sessions by **init**(8) and **inetd** (8C). Sessions are also created for processes that disassociate themselves from a controlling terminal using

ioctl(fd, TIOCNOTTY, 0)

or

setpgrp(mypid, 0) For more information about sessions, see **setsid**(2V).

Signal

Signals are used for notification of asynchronous events. Signals may directed to processes, process groups, and other combinations of processes. Signals may be sent by a process or by the operating system. Some signals may be caught. There is typically a default behavior on receipt if they are not caught. For more information about signals, see **signal**(3V), **kill**(2V), **sigvec**(2), **termio**(4).

Sockets and Address Families

A socket is an endpoint for communication between processes, similar to the way a telephone is the endpoint of communication between humans. Each socket has queues for sending and receiving data.

Sockets are typed according to their communications properties. These properties include whether messages sent and received at a socket require the name of the partner, whether communication is reliable, the format used in naming message recipients, etc.

Each instance of the system supports some collection of socket types; consult **socket**(2) for more information about the types available and their properties.

Each instance of the system supports some number of sets of communications protocols. Each protocol set supports addresses of a certain format. An Address Family is the set of addresses for a specific group of protocols. Each socket has an address chosen from the address family in which the socket was created.

Special Processes

The processes with a process ID's of 0, 1, and 2 are special. Process 0 is the scheduler. Process 1 is the initialization process **init**, and is the ancestor of every other process in the system. It is used to control the process structure. Process 2 is the paging daemon.

Super-user

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

Tty Process Group

Each active process can be a member of a terminal group that is identified by a positive integer called the tty process group ID. This grouping is used to arbitrate between multiple jobs contending for the same terminal (see **csh**(1), and **termio**(4)), to direct signals (tty and job control) to the appropriate process group, and to terminate a group of related processes upon termination of one of the processes in the group (see **exit**(2V) and **sigvec**(2)).

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 *emulator*, and need not be 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.

SYSTEM V IPC

The SunOS system supports the System V IPC namespace. For information about shared memory, semaphores and messages see **msgctl**(2), **msgget**(2), **msgop**(2), **semctl**(2), **semget**(2), **semop**(2), **shmctl**(2), **shmget**(2) and **shmop**(2).

ERROR CODES

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 given in **<errno.h>**.

E2BIG 7 Arg list too long
: An argument list longer than 1,048,576 bytes is presented to **execve**(2V) or a routine that called **execve**( ).

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

EADDRINUSE 48 Address already in use
: Only one usage of each address is normally permitted.

EADDRNOTAVAIL 49 Can't assign requested address
: Normally results from an attempt to create a socket with an address not on this machine.

EADV 83 Advertise error
: An attempt was made to advertise a resource which has been advertised already, or to stop the RFS while there are resources still advertised, or to force unmount a resource when it is still advertised. This error is RFS specific.

EAFNOSUPPORT 47 Address family not supported by protocol family
: An address incompatible with the requested protocol was used. For example, you should not necessarily expect to be able to use PUP Internet addresses with ARPA Internet protocols.

EAGAIN 11 No more processes
: A **fork**(2V) 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 resources.

EALREADY 37 Operation already in progress
: An operation was attempted on a non-blocking object that already had an operation in progress.

EBADF 9 Bad file number
: Either a file descriptor refers to no open file, or a read (respectively, write) request is made to a file that is open only for writing (respectively, reading).

EBADMSG 76 Not a data message
: During a **read**(2V), **getmsg**(2), or **ioctl**(2) I_RECVFD system call to a STREAMS device, something has come to the head of the queue that cannot be processed. That something depends on the system call

read(2V)	control information or a passed file descriptor.
getmsg(2)	passed file descriptor.
ioctl(2)	control or data information.

EBUSY 16 Device busy
: An attempt was made to mount a file system that was already mounted or an attempt was made to dismount a file system on which there is an active file (open file, mapped file, current directory, or mounted-on directory).

ECHILD 10 No children
: A **wait**(2V) was executed by a process that had no existing or unwaited-for child processes.

ECOMM 85 Communication error on send
: An attempt was made to send messages to a remote machine when no virtual circuit could be found. This error is RFS specific.

ECONNABORTED 53 Software caused connection abort
: A connection abort was caused internal to your host machine.

ECONNREFUSED 61 Connection refused
: No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host.

ECONNRESET 54 Connection reset by peer
A connection was forcibly closed by a peer. This normally results from the peer executing a **shutdown**(2) call.

EDEADLK 78 Deadlock situation detected/avoided
An attempt was made to lock a system resource that would have resulted in a deadlock situation.

EDESTADDRREQ 39 Destination address required
A required address was omitted from an operation on a socket.

EDOM 33 Math argument
The argument of a function in the math library (as described in section 3M) is out of the domain of the function.

EDQUOT 69 Disc quota exceeded
A **write()** to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because the user's quota of disk blocks was exhausted, or the allocation of an inode for a newly created file failed because the user's quota of inodes was exhausted.

EEXIST 17 File exists
An existing file was mentioned in an inappropriate context, for example, **link**(2V).

EFAULT 14 Bad address
The system encountered a hardware fault in attempting to access the arguments of a system call.

EFBIG 27 File too large
The size of a file exceeded the maximum file size (1,082,201,088 bytes).

EHOSTDOWN 64 Host is down
A socket operation failed because the destination host was down.

EHOSTUNREACH 65 Host is unreachable
A socket operation was attempted to an unreachable host.

EIDRM 77 Identifier removed
This error is returned to processes that resume execution due to the removal of an identifier.

EINPROGRESS 36 Operation now in progress
An operation that takes a long time to complete (such as a **connect**(2)) was attempted on a non-blocking object (see **ioctl**(2)).

EINTR 4 Interrupted system call
An asynchronous signal (such as interrupt or quit) that the process has elected to catch occurred during a system call. If execution is resumed after processing the signal, and the system call is not restarted, it will appear as if the interrupted system call returned this error condition.

EINVAL 22 Invalid argument
A system call was made with an invalid argument; for example, dismounting a non-mounted file system, mentioning an unknown signal in **sigvec()** or **kill()**, reading or writing a file for which **lseek()** has generated a negative pointer, or some other argument inappropriate for the call. Also set by math functions, see **intro**(3).

EIO 5 I/O error
Some physical I/O error occurred. This error may in some cases occur on a call following the one to which it actually applies.

EISCONN 56 Socket is already connected
A **connect()** request was made on an already connected socket; or, a **sendto()** or **sendmsg()** request on a connected socket specified a destination other than the connected party.

EISDIR 21 Is a directory
An attempt was made to write on a directory.

EISDIR 21 Is a directory
An attempt was made to write on a directory.

ELOOP 62 Too many levels of symbolic links
A path name lookup involved more than 20 symbolic links.

EMFILE 24 Too many open files
A process tried to have more open files than the system allows a process to have. The customary configuration limit is 64 per process.

EMLINK 31 Too many links
An attempt was made to make more than 32767 hard links to a file.

EMSGSIZE 40 Message too long
A message sent on a socket was larger than the internal message buffer.

EMULTIHOP 87 Multihop attempted
An attempt was made to access remote resources which are not directly accessible. This error is RFS specific.

ENAMETOOLONG 63 File name too long
A component of a path name exceeded 255 characters, or an entire path name exceeded 1024 characters.

ENETDOWN 50 Network is down
A socket operation encountered a dead network.

ENETRESET 52 Network dropped connection on reset
The host you were connected to crashed and rebooted.

ENETUNREACH 51 Network is unreachable
A socket operation was attempted to an unreachable network.

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

ENOBUFS 55 No buffer space available
An operation on a socket or pipe was not performed because the system lacked sufficient buffer space.

ENODEV 19 No such device
An attempt was made to apply an inappropriate system call to a device (for example, an attempt to read a write-only device) or an attempt was made to use a device not configured by the system.

ENOENT 2 No such file or directory
This error occurs when a file name is specified and the file should exist but does not, or when one of the directories in a path name does not exist.

ENOEXEC 8 Exec format error
A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number (see **a.out**(5)).

ENOLCK 79 No locks available
A system-imposed limit on the number of simultaneous file and record locks was reached and no more were available at that time.

ENOLINK 82 Link has be severed
The link (virtual circuit) connecting to a remote machine is gone. This error is RFS specific.

ENOMEM 12 Not enough memory
: During an **execve**(2V), **sbrk**( ), or **brk**(2), a program asks for more address space or swap space than the system is able to supply, or a process size limit would be exceeded. A lack of swap space is normally a temporary condition; however, a lack of address space is not a temporary condition. The maximum size of the text, data, and stack segments is a system parameter. Soft limits may be increased to their corresponding hard limits.

ENOMSG 75 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).

ENONET 80 Machine is not on the network
: A attempt was made to advertise, unadvertise, mount, or unmount remote resources while the machine has not done the proper startup to connect to the network. This error is Remote File Sharing (RFS) specific.

ENOPROTOOPT 42 Option not supported by protocol
: A bad option was specified in a **setsockopt**( ) or **getsockopt**(2) call.

ENOSPC 28 No space left on device
: A **write**( ) to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because no more disk blocks are available on the file system, or the allocation of an inode for a newly created file failed because no more inodes are available on the file system.

ENOSR 74 Out of stream resources
: During a STREAMS **open**(2V), either no STREAMS queues or no STREAMS head data structures were available.

ENOSTR 72 Not a stream device
: A **putmsg**(2) or **getmsg**(2) system call was attempted on a file descriptor that is not a STREAMS device.

ENOSYS 90 Function not implemented
: An attempt was made to use a function that is not available in this implementation.

ENOTBLK 15 Block device required
: A file that is not a block device was mentioned where a block device was required, for example, in **mount**(2V).

ENOTCONN 57 Socket is not connected
: An request to send or receive data was disallowed because the socket is not connected.

ENOTDIR 20 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**(2V).

ENOTEMPTY 66 Directory not empty
: An attempt was made to remove a directory with entries other than '**&.**' and '**&.|.**' by performing a **rmdir**( ) system call or a **rename**( ) system call with that directory specified as the target directory.

ENOTSOCK 38 Socket operation on non-socket
: Self-explanatory.

ENOTTY 25 Inappropriate ioctl for device
: The code used in an **ioctl**( ) call is not supported by the object that the file descriptor in the call refers to.

ENXIO 6 No such device or address
: I/O on a special file refers to a subdevice that 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.

EOPNOTSUPP 45 Operation not supported on socket
For example, trying to *accept* a connection on a datagram socket.

EPERM 1 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.

EPFNOSUPPORT 46 Protocol family not supported
The protocol family has not been configured into the system or no implementation for it exists.

EPIPE 32 Broken pipe
An attempt was made to write on a pipe or socket for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is caught or ignored.

EPROTO 86 Protocol error
Some protocol error occurred. This error is device specific, but is generally not related to a hardware failure.

EPROTONOSUPPORT 43 Protocol not supported
The protocol has not been configured into the system or no implementation for it exists.

EPROTOTYPE 41 Protocol wrong type for socket
A protocol was specified that does not support the semantics of the socket type requested. For example, you cannot use the ARPA Internet UDP protocol with type **SOCK_STREAM**.

ERANGE 34 Result too large
The value of a function in the math library (as described in section 3M) is unrepresentable within machine precision.

EREMOTE 71 Too many levels of remote in path
An attempt was made to remotely mount a file system into a path that already has a remotely mounted component.

EROFS 30 Read-only file system
An attempt to modify a file or directory was made on a file system mounted read-only.

ERREMOTE 81 Object is remote
An attempte was made to advertise a resource which is not on the local machine, or to mount/unmount a device (or pathname) that is on a remote machine. This error is RFS specific.

ESHUTDOWN 58 Can't send after socket shutdown
A request to send data was disallowed because the socket had already been shut down with a previous **shutdown**(2) call.

ESOCKTNOSUPPORT 44 Socket type not supported
The support for the socket type has not been configured into the system or no implementation for it exists.

ESPIPE 29 Illegal seek
An **lseek**( ) was issued to a socket or pipe. This error may also be issued for other non-seekable devices.

ESRCH 3 No such process
The process or process group whose number was given does not exist, or any such process is already dead.

ESRMNT 84 Srmount error
An attempt was made to stop RFS while there are resources still mounted by remote machines. This error is RFS specific.

ESTALE 70 Stale NFS file handle
An NFS client referenced a file that it had opened but that had since been deleted.

ETIME 73 Timer expired
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.

ETIMEDOUT 60 Connection timed out
A *connect* request or an NFS request failed because the party to which the request was made did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.)

ETXTBSY 26 Text file busy
An attempt was made to execute a pure-procedure program that is currently open for writing, or an attempt was made to open for writing a pure-procedure program that is being executed.

EUSERS 68 Too many users
An operation to read disk quota information for the user failed because the system quota table was full.

EWOULDBLOCK 35 Operation would block
An operation that would cause a process to block was attempted on an object in non-blocking mode (see **ioctl**(2)).

EXDEV 18 Cross-device link
A hard link to a file on another file system was attempted.

unused 0

SEE ALSO

brk(2), **chdir**(2V), **chmod**(2V), **connect**(2), **dup**(2V), **execve**(2V), **exit**(2V), **fork**(2V), **getmsg**(2), **getsockopt**(2), **ioctl**(2), **killpg**(2), **link**(2V), **mount**(2V), **msgctl**(2), **msgget**(2), **msgop**(2), **open**(2V), **pipe**(2V), **putmsg**(2), **read**(2V), **semctl**(2), **semget**(2), **semop**(2), **getsockopt**(2), **shmctl**(2), **shmget**(2), **shmop**(2), **shutdown**(2), **sigvec**(2), **socket**(2), **socketpair**(2), **wait**(2V), **csh**(1), **sh**(1), **intro**(3), **perror**(3) **termio**(4), **a.out**(5)

LIST OF SYSTEM CALLS

Name	Appears on Page	Description
accept	**accept**(2)	accept a connection on a socket
access	**access**(2V)	determine accessibility of file
acct	**acct**(2V)	turn accounting on or off
adjtime	**adjtime**(2)	correct the time to allow synchronization of the system clock
async_daemon	**nfssvc**(2)	NFS daemons
audit	**audit**(2)	write a record to the audit log
auditon	**auditon**(2)	manipulate auditing
auditsvc	**auditsvc**(2)	write audit records to specified file descriptor
bind	**bind**(2)	bind a name to a socket
brk	**brk**(2)	change data segment size
chdir	**chdir**(2V)	change current working directory
chmod	**chmod**(2V)	change mode of file
chown	**chown**(2V)	change owner and group of a file
chroot	**chroot**(2)	change root directory
close	**close**(2V)	delete a descriptor
connect	**connect**(2)	initiate a connection on a socket
creat	**creat**(2V)	create a new file
dup	**dup**(2V)	duplicate a descriptor
dup2	**dup**(2V)	duplicate a descriptor

execve	**execve**(2V)	execute a file
_exit	**exit**(2V)	terminate a process
fchmod	**chmod**(2V)	change mode of file
fchown	**chown**(2V)	change owner and group of a file
fcntl	**fcntl**(2V)	file control
flock	**flock**(2)	apply or remove an advisory lock on an open file
fork	**fork**(2V)	create a new process
fpathconf	**pathconf**(2V)	query file system related limits and options
fstat	**stat**(2V)	get file status
fstatfs	**statfs**(2)	get file system statistics
fsync	**fsync**(2)	synchronize a file's in-core state with that on disk
ftruncate	**truncate**(2)	set a file to a specified length
getauid	**getauid**(2)	get and set user audit identity
getdents	**getdents**(2)	gets directory entries in a filesystem independent format
getdirentries	**getdirentries**(2)	gets directory entries in a filesystem independent format
getdomainname	**getdomainname**(2)	get/set name of current domain
getdtablesize	**getdtablesize**(2)	get descriptor table size
getegid	**getgid**(2V)	get group identity
geteuid	**getuid**(2V)	get user identity
getgid	**getgid**(2V)	get group identity
getgroups	**getgroups**(2V)	get or set supplementary group IDs
gethostid	**gethostid**(2)	get unique identifier of current host
gethostname	**gethostname**(2)	get/set name of current host
getitimer	**getitimer**(2)	get/set value of interval timer
getmsg	**getmsg**(2)	get next message from a stream
getpagesize	**getpagesize**(2)	get system page size
getpeername	**getpeername**(2)	get name of connected peer
getpgrp	**getpgrp**(2V)	return or set the process group of a process
getpid	**getpid**(2V)	get process identification
getppid	**getpid**(2V)	get process identification
getpriority	**getpriority**(2)	get/set process nice value
getrlimit	**getrlimit**(2)	control maximum system resource consumption
getrusage	**getrusage**(2)	get information about resource utilization
getsockname	**getsockname**(2)	get socket name
getsockopt	**getsockopt**(2)	get and set options on sockets
gettimeofday	**gettimeofday**(2)	get or set the date and time
getuid	**getuid**(2V)	get user identity
ioctl	**ioctl**(2)	control device
kill	**kill**(2V)	send a signal to a process or a group of processes
killpg	**killpg**(2)	send signal to a process group
link	**link**(2V)	make a hard link to a file
listen	**listen**(2)	listen for connections on a socket
lseek	**lseek**(2V)	move read/write pointer
lstat	**stat**(2V)	get file status
mctl	**mctl**(2)	memory management control
mincore	**mincore**(2)	determine residency of memory pages
mkdir	**mkdir**(2V)	make a directory file
mkfifo	**mknod**(2V)	make a special file
mknod	**mknod**(2V)	make a special file
mmap	**mmap**(2)	map pages of memory
mount	**mount**(2V)	mount file system
mprotect	**mprotect**(2)	set protection of memory mapping
msgctl	**msgctl**(2)	message control operations

msgget	**msgget**(2)	get message queue
msgop	**msgop**(2)	message operations
msgrcv	**msgop**(2)	message operations
msgsnd	**msgop**(2)	message operations
msync	**msync**(2)	synchronize memory with physical storage
munmap	**munmap**(2)	unmap pages of memory.
nfssvc	**nfssvc**(2)	NFS daemons
open	**open**(2V)	open or create a file for reading or writing
pathconf	**pathconf**(2V)	query file system related limits and options
pipe	**pipe**(2V)	create an interprocess communication channel
poll	**poll**(2)	I/O multiplexing
profil	**profil**(2)	execution time profile
ptrace	**ptrace**(2)	process trace
putmsg	**putmsg**(2)	send a message on a stream
quotactl	**quotactl**(2)	manipulate disk quotas
read	**read**(2V)	read input
readlink	**readlink**(2)	read value of a symbolic link
readv	**read**(2V)	read input
reboot	**reboot**(2)	reboot system or halt processor
recv	**recv**(2)	receive a message from a socket
recvfrom	**recv**(2)	receive a message from a socket
recvmsg	**recv**(2)	receive a message from a socket
rename	**rename**(2V)	change the name of a file
rmdir	**rmdir**(2V)	remove a directory file
sbrk	**brk**(2)	change data segment size
select	**select**(2)	synchronous I/O multiplexing
semctl	**semctl**(2)	semaphore control operations
semget	**semget**(2)	get set of semaphores
semop	**semop**(2)	semaphore operations
send	**send**(2)	send a message from a socket
sendmsg	**send**(2)	send a message from a socket
sendto	**send**(2)	send a message from a socket
setaudit	**setuseraudit**(2)	set the audit classes for a specified user ID
setauid	**getauid**(2)	get and set user audit identity
setdomainname	**getdomainname**(2)	get/set name of current domain
setgroups	**getgroups**(2V)	get or set supplementary group IDs
sethostname	**gethostname**(2)	get/set name of current host
setitimer	**getitimer**(2)	get/set value of interval timer
setpgid	**setpgid**(2V)	set process group ID for job control
setpgrp	**getpgrp**(2V)	return or set the process group of a process
setpriority	**getpriority**(2)	get/set process nice value
setregid	**setregid**(2)	set real and effective group IDs
setreuid	**setreuid**(2)	set real and effective user IDs
setrlimit	**getrlimit**(2)	control maximum system resource consumption
setsid	**setsid**(2V)	create session and set process group ID
setsockopt	**getsockopt**(2)	get and set options on sockets
settimeofday	**gettimeofday**(2)	get or set the date and time
setuseraudit	**setuseraudit**(2)	set the audit classes for a specified user ID
sgetl	**sputl**(2)	access long integer data in a machine-independent fashion
shmat	**shmop**(2)	shared memory operations
shmctl	**shmctl**(2)	shared memory control operations
shmdt	**shmop**(2)	shared memory operations
shmget	**shmget**(2)	get shared memory segment identifier

shmop	**shmop**(2)	shared memory operations
shutdown	**shutdown**(2)	shut down part of a full-duplex connection
sigblock	**sigblock**(2)	block signals
sigmask	**sigblock**(2)	block signals
sigpause	**sigpause**(2V)	automatically release blocked signals and wait for interrupt
sigpending	**sigpending**(2V)	examine pending signals
sigprocmask	**sigprocmask**(2V)	examine and change blocked signals
sigsetmask	**sigsetmask**(2)	set current signal mask
sigstack	**sigstack**(2)	set and/or get signal stack context
sigsuspend	**sigpause**(2V)	automatically release blocked signals and wait for interrupt
sigvec	**sigvec**(2)	software signal facilities
socket	**socket**(2)	create an endpoint for communication
socketpair	**socketpair**(2)	create a pair of connected sockets
sputl	**sputl**(2)	access long integer data in a machine-independent fashion
stat	**stat**(2V)	get file status
statfs	**statfs**(2)	get file system statistics
swapon	**swapon**(2)	add a swap device for interleaved paging/swapping
symlink	**symlink**(2)	make symbolic link to a file
sync	**sync**(2)	update super-block
syscall	**syscall**(2)	indirect system call
sysconf	**sysconf**(2V)	query system related limits, values, options
tell	**lseek**(2V)	move read/write pointer
truncate	**truncate**(2)	set a file to a specified length
umask	**umask**(2V)	set file creation mode mask
umount	**unmount**(2V)	remove a file system
uname	**uname**(2V)	get information about current system
unlink	**unlink**(2V)	remove directory entry
unmount	**unmount**(2V)	remove a file system
ustat	**ustat**(2)	get file system statistics
utimes	**utimes**(2)	set file times
vadvise	**vadvise**(2)	give advice to paging system
vfork	**vfork**(2)	spawn new process in a virtual memory efficient way
vhangup	**vhangup**(2)	virtually ''hangup'' the current control terminal
wait	**wait**(2V)	wait for process to terminate or stop, examine returned status
wait3	**wait**(2V)	wait for process to terminate or stop, examine returned status
wait4	**wait**(2V)	wait for process to terminate or stop, examine returned status
waitpid	**wait**(2V)	wait for process to terminate or stop, examine returned status
WEXITSTATUS	**wait**(2V)	wait for process to terminate or stop, examine returned status
WIFEXITED	**wait**(2V)	wait for process to terminate or stop, examine returned status
WIFSIGNALED	**wait**(2V)	wait for process to terminate or stop, examine returned status
WIFSTOPPED	**wait**(2V)	wait for process to terminate or stop, examine returned status
write	**write**(2V)	write output
writev	**write**(2V)	write output
WSTOPSIG	**wait**(2V)	wait for process to terminate or stop, examine returned status
WTERMSIG	**wait**(2V)	wait for process to terminate or stop, examine returned status

NAME

accept – accept a connection on a socket

SYNOPSIS

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

int accept(s, addr, addrlen)
int s;
struct sockaddr *addr;
int *addrlen;
```

DESCRIPTION

The argument *s* is a socket that has been created with **socket**(2), bound to an address with **bind**(2), and is listening for connections after a **listen**(2). **accept**( ) extracts the first connection on the queue of pending connections, creates a new socket with the same properties of *s* and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, **accept**( ) blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, **accept**( ) returns an error as described below. The accepted socket is used to read and write data to and from the socket which connected to this one; it is not used to accept more connections. The original socket *s* remains open for accepting further connections.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication is occurring. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with SOCK_STREAM.

It is possible to **select**(2) a socket for the purposes of doing an **accept**( ) by selecting it for read.

RETURN VALUES

accept( ) returns a non-negative descriptor for the accepted socket on success. On failure, it returns –1 and sets **errno** to indicate the error.

ERRORS

EBADF	The descriptor is invalid.
EFAULT	The *addr* parameter is not in a writable part of the user address space.
ENOTSOCK	The descriptor references a file, not a socket.
EOPNOTSUPP	The referenced socket is not of type SOCK_STREAM.
EWOULDBLOCK	The socket is marked non-blocking and no connections are present to be accepted.

SEE ALSO

bind(2), **connect**(2), **listen**(2), **select**(2), **socket**(2)

NAME

access – determine accessibility of file

SYNOPSIS

```
#include <unistd.h>

int access(path, mode)
char *path;
int mode;
```

DESCRIPTION

path points to a path name naming a file. **access()** checks the named file for accessibility according to *mode*, which is an inclusive or of the following bits:

R_OK	test for read permission
W_OK	test for write permission
X_OK	test for execute or search permission

The following value may also be supplied for *mode*:

F_OK	test whether the directories leading to the file can be searched and the file exists.

The real user ID and the supplementary group IDs (including the real group ID) are used in verifying permission, so this call is useful to set-UID programs.

Notice that only access bits are checked. A directory may be indicated as writable by **access()**, but an attempt to open it for writing will fail (although files may be created there); a file may look executable, but **execve()** will fail unless it is in proper format.

RETURN VALUES

access() returns:

0	on success.
−1	on failure and sets **errno** to indicate the error.

ERRORS

EACCES	Search permission is denied for a component of the path prefix of *path*.
	The file access permissions do not permit the requested access to the file named by *path*.
EFAULT	*path* points outside the process's allocated address space.
EINVAL	An invalid value was specified for *mode*.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	The file named by *path* does not exist.
ENOTDIR	A component of the path prefix of *path* is not a directory.
EROFS	The file named by *path* is on a read-only file system and write access was requested.

SYSTEM V ERRORS

In addtion to the above, the following may also occur:

ENOENT	*path* points to an empty string.

SEE ALSO

chmod(2V), **stat**(2V)

NAME

acct – turn accounting on or off

SYNOPSIS

int acct (path)
char *path;

DESCRIPTION

acct() is used to enable or disable the process accounting. If process accounting 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 (2V)** and **sigvec(2)**. The effective user ID of the calling process must be super-user to use this call.

path points to a path name naming the accounting file. The accounting file format is given in **acct**(5).

The accounting routine is enabled if *path* is not a NULL pointer and no errors occur during the system call. It is disabled if *path* is a NULL pointer and no errors occur during the system call.

If accounting is already turned on, and a successful **acct()** call is made with a non-NULL *path*, all subsequent accounting records will be written to the new accounting file.

SYSTEM V DESCRIPTION

If accounting is already turned on, it is an error to call **acct()** with a non-NULL path.

RETURN VALUES

acct() returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

EACCES	Search permission is denied for a component of the path prefix of *path*.
	The file referred to by *path* is not a regular file.
EFAULT	*path* points outside the process's allocated address space.
EINVAL	Support for accounting was not configured into the system.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating the path name.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} (see **sysconf**(2V)) while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	The named file does not exist.
ENOTDIR	A component of the path prefix of *path* is not a directory.
EPERM	The caller is not the super-user.
EROFS	The named file resides on a read-only file system.

SYSTEM V ERRORS

EBUSY	*path* is non-NULL, and accounting is already turned on.
ENOENT	*path* points to an empty string.

SEE ALSO

exit(2V), **sigvec**(2), **acct**(5), **sa**(8)

BUGS

No accounting records are produced for programs running when a crash occurs. In particular non-terminating programs are never accounted for.

NOTES

Accounting is automatically disabled when free space on the file system the accounting file resides on drops below 2 percent; it is enabled when free space rises above 4 percent.

NAME

adjtime – correct the time to allow synchronization of the system clock

SYNOPSIS

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

int adjtime(delta, olddelta)
struct timeval *delta;
struct timeval *olddelta;
```

DESCRIPTION

adjtime( ) adjusts the system's notion of the current time, as returned by **gettimeofday**(2), advancing or retarding it by the amount of time specified in the **struct timeval** (defined in **<sys/time.h>**) pointed to by *delta*.

The adjustment is effected by speeding up (if that amount of time is positive) or slowing down (if that amount of time is negative) the system's clock by some small percentage, generally a fraction of one percent. Thus, the time is always a monotonically increasing function. A time correction from an earlier call to **adjtime( )** may not be finished when **adjtime( )** is called again. If *olddelta* is not a NULL pointer, then the structure it points to will contain, upon return, the number of microseconds still to be corrected from the earlier call. If *olddelta* is a NULL pointer, the corresponding information will not be returned.

This call may be used in time servers that synchronize the clocks of computers in a local area network. Such time servers would slow down the clocks of some machines and speed up the clocks of others to bring them to the average network time.

Only the super-user may adjust the time of day.

The adjustment value will be silently rounded to the resolution of the system clock.

RETURN

A 0 return value indicates that the call succeeded. A –1 return value indicates an error occurred, and in this case an error code is stored into the global variable **errno**.

ERRORS

EFAULT	*delta* or *olddelta* points outside the process's allocated address space.
	olddelta points to a region of the process' allocated address space that is not writable.
EPERM	The process's effective user ID is not that of the super-user.

SEE ALSO

date(1V), **gettimeofday**(2)

NAME

audit – write a record to the audit log

SYNOPSIS

```
#include <sys/label.h>
#include <sys/audit.h>

int audit (record)
audit_record_t *record;
```

DESCRIPTION

The **audit( )** system call is used to write a record to the system audit log file. The data pointed to by *record* is written to the audit log file. The data should be a well-formed audit record as described by **audit.log**(5). The kernel sets the time stamp value in the record and performs a minimal check on the data before writing it to the audit log file.

Only the super-user may successfully execute this call.

RETURN VALUES

audit( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

EFAULT	*record* points outside the process's allocated address space.
EINVAL	The length specified in the audit record is too short, or more than MAXAUDITDATA.
EPERM	The process's effective user ID is not super-user.

SEE ALSO

auditsvc(2), **getauid**(2), **setuseraudit**(2), **audit_args**(3), **audit.log**(5), **auditd**(8)

NAME

auditon – manipulate auditing

SYNOPSIS

```
#include <sys/label.h>
#include <sys/audit.h>

int auditon (condition)
int condition;
```

DESCRIPTION

The **auditon()** system call sets system auditing to the requested *condition* if and only if the current state of auditing allows that transition. Legitimate values for *condition* are:

AUC_UNSET	on/off has not been decided yet
AUC_AUDITING	auditing is to be done
AUC_NOAUDIT	auditing is not to be done

The permitted transitions are:

- Any condition may be changed back to itself.
- AUC_UNSET may be changed to AUC_AUDITING or AUC_NOAUDIT.
- AUC_AUDITING may be changed to AUC_NOAUDIT.
- AUC_NOAUDIT may be changed to AUC_AUDITING.

Once changed, it is not possible to get back to AUC_UNSET.

Only the super-user may successfully execute this call.

RETURN VALUES

auditon() returns the old audit condition value on success. On failure, it returns –1 and sets **errno** to indicate the error.

ERRORS

EINVAL	The *condition* specified is outside the range of valid values.
	The current condition precludes the requested change.
EPERM	Neither of the process's effective or real user ID is super-user.

SEE ALSO

audit(2), **setuseraudit**(2)

NAME

auditsvc – write audit records to specified file descriptor

SYNOPSIS

int auditsvc(fd, limit)
int fd;
int limit;

DESCRIPTION

The **auditsvc()** system call specifies the audit log file to the kernel. The kernel writes audit records to this file until an exceptional condition occurs and then the call returns. The parameter *fd* is a file descriptor that identifies the audit file. Programs should open this file for writing before calling **auditsvc()**. The parameter *limit* specifies a value between 0 and 100, instructing **auditsvc()** to return when the percentage of free disk space on the audit filesystem drops below this limit. Thus, the invoking program can take action to avoid running out of disk space. The **auditsvc()** system call does not return until one of the following conditions occurs:

- The process receives a signal that is not blocked or ignored.
- An error is encountered writing to the audit log file.
- The minimum free space (as specified by *limit*), has been reached.

Only processes with a real or effective user ID of super-user may execute this call successfully.

RETURN VALUES

auditsvc() returns only on an error.

ERRORS

EAGAIN	The descriptor referred to a *stream*, was marked for System V-style non-blocking I/O, and no data could be written immediately.
EBADF	*fd* is not a valid descriptor open for writing.
EBUSY	A second process attempted to perform this call.
	A second process attempted to perform this call.
EDQUOT	The user's quota of disk blocks on the file system containing the file has been exhausted.
	Audit filesystem space is below the specified limit.
EFBIG	An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.
EINTR	The call is forced to terminate prematurely due to the arrival of a signal whose SV_INTERRUPT bit in **sv_flags** is set (see **sigvec**(2)). **signal**(3V), in the System V compatibility library, sets this bit for any signal it catches.
EINVAL	Auditing is disabled (see **auditon**(2)).
	fd does not refer to a file of an appropriate type. Regular files are always appropriate.
EIO	An I/O error occurred while reading from or writing to the file system.
ENOSPC	There is no free space remaining on the file system containing the file.
ENXIO	A hangup occurred on the *stream* being written to.
EPERM	The process's effective or real user ID is not super-user.
EWOULDBLOCK	The file was marked for 4.2BSD-style non-blocking I/O, and no data could be written immediately.

SEE ALSO

audit(2), **auditon**(2), **sigvec**(2), **signal**(3V), **audit.log**(5), **auditd**(8)

NAME

bind – bind a name to a socket

SYNOPSIS

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

int bind(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

DESCRIPTION

bind() assigns a name to an unnamed socket. When a socket is created with **socket**(2) it exists in a name space (address family) but has no name assigned. **bind()** requests that the name pointed to by *name* be assigned to the socket.

RETURN VALUES

bind() returns:

0	on success.
−1	on failure and sets **errno** to indicate the error.

ERRORS

EACCES	The requested address is protected, and the current user has inadequate permission to access it.
EADDRINUSE	The specified address is already in use.
EADDRNOTAVAIL	The specified address is not available from the local machine.
EBADF	*s* is not a valid descriptor.
EFAULT	The *name* parameter is not in a valid part of the user address space.
EINVAL	*namelen* is not the size of a valid address for the specified address family.
	The socket is already bound to an address.
ENOTSOCK	*s* is a descriptor for a file, not a socket.

The following errors are specific to binding names in the UNIX domain:

EACCES	Search permission is denied for a component of the path prefix of the path name in *name*.
EIO	An I/O error occurred while making the directory entry or allocating the inode.
EISDIR	A null path name was specified.
ELOOP	Too many symbolic links were encountered in translating the path name in *name*.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} (see **sysconf**(2V)) while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	A component of the path prefix of the path name in *name* does not exist.
ENOTDIR	A component of the path prefix of the path name in *name* is not a directory.
EROFS	The inode would reside on a read-only file system.

SEE ALSO

connect(2), **getsockname**(2), **listen**(2), **socket**(2), **unlink**(2V)

NOTES

Binding a name in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed (using **unlink**(2V),

The rules used in name binding vary between communication domains. Consult the manual entries in section 4 for detailed information.

NAME

brk, sbrk – change data segment size

SYNOPSIS

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

int brk(addr)
caddr_t addr;

caddr_t sbrk(incr)
int incr;
```

DESCRIPTION

brk( ) sets the system's idea of the lowest data segment location not used by the program (called the *break*) to *addr* (rounded up to the next multiple of the system's page size).

In the alternate function **sbrk( )**, *incr* more bytes are added to the program's data space and a pointer to the start of the new area is returned.

When a program begins execution using **execve( )** the break is set at the highest location defined by the program and data storage areas.

The **getrlimit**(2) system call may be used to determine the maximum permissible size of the *data* segment; it will not be possible to set the break beyond the **rlim_max** value returned from a call to **getrlimit( )**, that is to say, "**etext** + **rlim.rlim_max**." (See **end**(3) for the definition of **etext( )**.)

RETURN VALUES

brk( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

sbrk( ) returns the old break value on success. On failure, it returns (**caddr_t**) −1 and sets **errno** to indicate the error.

ERRORS

brk( ) and **sbrk( )** will fail and no additional memory will be allocated if one of the following occurs:

ENOMEM The data segment size limit, as set by **setrlimit( )** (see **getrlimit**(2)), would be exceeded.

The maximum possible size of a data segment (compiled into the system) would be exceeded.

Insufficient space exists in the swap area to support the expansion.

Out of address space; the new break value would extend into an area of the address space defined by some previously established mapping (see **mmap**(2)).

SEE ALSO

execve(2V), **mmap**(2), **getrlimit**(2), **malloc**(3V), **end**(3)

WARNINGS

Programs combining the **brk( )** and **sbrk( )** system calls and **malloc( )** will not work. Many library routines use **malloc( )** internally, so use **brk( )** and **sbrk( )** only when you know that **malloc( )** definitely will not be used by any library routine.

BUGS

Setting the break may fail due to a temporary lack of swap space. It is not possible to distinguish this from a failure caused by exceeding the maximum size of the data segment without consulting **getrlimit( )**.

NAME

chdir – change current working directory

SYNOPSIS

```
int chdir(path)
char *path;

int fchdir(fd)
int fd;
```

DESCRIPTION

chdir() and **fchdir()** make the directory specified by *path* or *fd* the current working directory. Subsequent references to pathnames not starting with '/' are relative to the new current working directory.

In order for a directory to become the current directory, a process must have execute (search) access to the directory.

RETURN VALUES

chdir() returns:

0	on success.
−1	on failure and sets **errno** to indicate the error.

ERRORS

EACCES	Search permission is denied for a component of the pathname.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	The named directory does not exist.
ENOTDIR	A component of the pathname is not a directory.

SYSTEM V ERRORS

In addition to the above, the following may also occur:

ENOENT	*path* points to an empty string.

WARNINGS

fchdir() is provided as a performance enhancement and is guaranteed to fail under certain conditions. In particular, if auditing is active the call will never succeed, and EINVAL will be returned. Applications which use this system call must be coded to detect this failure and switch to using **chdir()** from that point on.

NAME

chmod, fchmod – change mode of file

SYNOPSIS

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

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

int fchmod(fd, mode)
int fd, mode;
```

DESCRIPTION

chmod() sets the mode of the file referred to by *path* or the descriptor *fd* according to *mode*. *mode* is the inclusive OR of the file mode bits (see **stat**(2V) for a description of these bits).

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 and the process attempts to set the set group ID bit on a file owned by a group which is not in its supplementary group IDs, the S_ISGID bit (set group ID on execution) is cleared.

If the S_ISVTX (sticky) bit is set on a directory, an unprivileged user may not delete or rename files of other users in that directory.

If a user other than the super-user writes to a file, the set user ID and set group ID bits are turned off. This makes the system somewhat more secure by protecting set-user-ID (set-group-ID) files from remaining set-user-ID (set-group-ID) if they are modified, at the expense of a degree of compatibility.

RETURN VALUES

chmod() returns:

0	on success.
−1	on failure and sets **errno** to indicate the error.

ERRORS

chmod() will fail and the file mode will be unchanged if:

EACCES	Search permission is denied for a component of the path prefix of *path*.
EFAULT	*path* points outside the process's allocated address space.
EINVAL	*fd* refers to a socket, not to a file.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	The file referred to by *path* does not exist.
ENOTDIR	A component of the path prefix of *path* is not a directory.
EPERM	The effective user ID does not match the owner of the file and the effective user ID is not the super-user.
EROFS	The file referred to by *path* resides on a read-only file system.

fchmod() will fail if:

EBADF	The descriptor is not valid.

EIO	An I/O error occurred while reading from or writing to the file system.
EPERM	The effective user ID does not match the owner of the file and the effective user ID is not the super-user.
EROFS	The file referred to by *fd* resides on a read-only file system.

SYSTEM V ERRORS

In addition to the above, the following may also occur:

ENOENT	*path* points to a null pathname.

SEE ALSO

chown(2V), **open**(2V), **stat**(2V), **sticky**(8)

BUGS

S_ISVTX, the "sticky bit", is a misnomer, and is overloaded to mean different things for different file types.

NAME

chown, fchown – change owner and group of a file

SYNOPSIS

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

int fchown(fd, owner, group)
int fd;
int owner;
int group;
```

SYSTEM V SYNOPSIS

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

int chown(path, owner, group)
char *path;
uid_t owner;
gid_t group;
```

DESCRIPTION

The file that is named by *path* or referenced by *fd* has its *owner* and *group* changed as specified. Only the super-user may change the owner of the file, because if users were able to give files away, they could defeat the file-space accounting procedures (see NOTES). The owner of the file may change the group to a group of which he is a member. The super-user may change the group arbitrarily.

fchown( ) is particularly useful when used in conjunction with the file locking primitives (see **flock**(2)).

If *owner* or *group* is specified as –1, the corresponding ID of the file is not changed.

If a process whose effective user ID is not super-user successfully changes the group ID of a file, the set-user-ID and set-group-ID bits of the file mode, **S_ISUID** and **S_ISGID** respectively (see **stat**(2V)), will be cleared.

If the final component of *path* is a symbolic link, the ownership and group of the symbolic link is changed, not the ownership and group of the file or directory to which it points.

RETURN VALUES

chown( ) and **fchown**( ) return:

0	on success.
–1	on failure and set **errno** to indicate the error.

ERRORS

chown( ) will fail and the file will be unchanged if:

EACCES	Search permission is denied for a component of the path prefix of *path*.
EFAULT	*path* points outside the process's allocated address space.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} (see **sysconf**(2V)) while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	The file referred to by *path* does not exist.
ENOTDIR	A component of the path prefix of *path* is not a directory.

EPERM	The user ID specified by *owner* is not the current owner ID of the file.
	The group ID specified by *group* is not the current group ID of the file and is not in the process' supplementary group IDs, and the effective user ID is not the super-user.
EROFS	The file referred to by *path* resides on a read-only file system.

fchown() will fail if:

EBADF	*fd* does not refer to a valid descriptor.
EINVAL	*fd* refers to a socket, not a file.
EIO	An I/O error occurred while reading from or writing to the file system.
EPERM	The user ID specified by *owner* is not the current owner ID of the file.
	The group ID specified by *group* is not the current group ID of the file and is not in the supplementary group IDs, and the effective user ID is not the super-user.
EROFS	The file referred to by *fd* resides on a read-only file system.

SYSTEM V ERRORS

In addition to the above, the following may also occur:

ENOENT	*path* points to an empty string.

SEE ALSO

chmod(2V), **flock**(2)

NOTES

For **chown()** to behave as described above, {_POSIX_CHOWN_RESTRICTED} must be in effect (see **pathconf**(2V)). {_POSIX_CHOWN_RESTRICTED} is always in effect on SunOS systems, but for portability, applications should call **pathconf()** to determine whether {_POSIX_CHOWN_RESTRICTED} is in effect for *path*.

If {_POSIX_CHOWN_RESTRICTED} is in effect for the file system on which the file referred to by *path* or *fd* resides, only the super-user may change the owner of the file. Otherwise, processes with effective user ID equal to the file owner or super-user may change the owner of the file.

NAME

chroot – change root directory

SYNOPSIS

int chroot(dirname)
char *dirname;

int fchroot(fd)
int fd;

DESCRIPTION

chroot( ) and **fchroot( )** cause a directory to become the root directory, the starting point for path names beginning with '/'. The current working directory is unaffected by this call. This root directory setting is inherited across **execve**(2V) and by all children of this process created with **fork** (2V) calls.

In order for a directory to become the root directory a process must have execute (search) access to the directory and either the effective user ID of the process must be super-user or the target directory must be the system root or a loop-back mount of the system root (see **lofs**(4S)). **fchroot( )** is further restricted in that while it is always possible to change to the system root using this call, it is not guaranteed to succeed in any other case, even should *fd* be in all respects valid.

The *dirname* argument to **chroot( )** points to a path name of a directory. The *fd* argument to **fchroot( )** is the open file descriptor of the directory which is to become the root.

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. Instead, **fchroot( )** can be used to set the root back to a directory which was opened before the root directory was changed.

WARNINGS

The only use of **fchroot( )** that is appropriate is to change back to the system root. While it may succeed in some other cases, it is guaranteed to fail if auditing is enabled. Super-user processes are not exempt from this limitation.

RETURN VALUES

chroot( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

chroot( ) will fail and the root directory will be unchanged if one or more of the following are true:

EACCES	Search permission is denied for a component of the path prefix of *dirname*.
	Search permission is denied for the directory referred to by *dirname*.
EBADF	The descriptor is not valid.
EFAULT	*dirname* points outside the process's allocated address space.
EINVAL	**fchroot( )** attempted to change to a directory which is not the system root and external circumstances, such as auditing, do not allow this.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *dirname*.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} (see **sysconf**(2V)) while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	The directory referred to by *dirname* does not exist.

ENOTDIR	A component of the path prefix of *dirname* is not a directory.
	The file referred to by *dirname* is not a directory.
EPERM	The effective user ID is not super-user.

SEE ALSO

chdir(2V), **execve**(2V), **fork**(2V), **lofs**(4S)

NAME

close – delete a descriptor

SYNOPSIS

int close (fd)
int fd;

DESCRIPTION

close( ) deletes a descriptor from the per-process object reference table. If *fd* is the last reference to the underlying object, then the object will be deactivated. For example, on the last close of a file the current *seek* pointer associated with the file is lost. On the last close of a socket (see **socket**(2)), associated naming information and queued data are discarded. On the last close of a file holding an advisory lock applied by **flock**(2), the lock is released. (Record locks applied to the file by **lockf**(3), however, are released on *any* call to **close( )** regardless of whether *fd* is the last reference to the underlying object.)

close( ) does not unmap any mapped pages of the object referred to by *fd* (see **mmap( )**, **munmap**(2)).

A close of all of a process's descriptors is automatic on **exit( )**, but since there is a limit on the number of active descriptors per process, **close( )** is necessary for programs that deal with many descriptors.

When a process forks (see **fork**(2v)), all descriptors for the new child process reference the same objects as they did in the parent before the fork. If a new process is then to be run using **execve**(2V), the process would normally inherit these descriptors. Most of the descriptors can be rearranged with **dup**(2V) or deleted with **close( )** before the **execve( )** is attempted, but if some of these descriptors will still be needed if the **execve( )** fails, it is necessary to arrange for them to be closed if the **execve( )** succeeds. The **fcntl**(2V) operation **F_SETFD** can be used to arrange that a descriptor will be closed after a successful **execve( )**, or to restore the default behavior, which is to not close the descriptor.

If a STREAMS (see **intro**(2)) file is closed, and the calling process had previously registered to receive a SIGPOLL signal (see **sigvec**(2)) for events associated with that file (see **I_SETSIG** in **streamio**(4)), the calling process will be unregistered for events associated with the file. The last **close( )** for a stream causes that stream to be dismantled. If the descriptor is not marked for no-delay mode 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 descriptor is marked for no-delay mode or if there are any pending signals, **close( )** does not wait for output to drain, and dismantles the stream immediately.

RETURN VALUES

close( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

EBADF *fd* is not an active descriptor.

EINTR A signal was caught before the close completed.

SEE ALSO

accept(2), **dup**(2V), **execve**(2V), **fcntl**(2V), **flock**(2), **intro**(2), **open**(2V), **pipe**(2V), **sigvec**(2), **socket**(2), **socketpair**(2), **streamio**(4)

NAME

connect – initiate a connection on a socket

SYNOPSIS

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

int connect(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

DESCRIPTION

The parameter *s* is a socket. If it is of type **SOCK_DGRAM**, then this call specifies the peer with which the socket is to be associated; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If it is of type **SOCK_STREAM**, then this call attempts to make a connection to another socket. The other socket is specified by *name* which is an address in the communications space of the socket. Each communications space interprets the *name* parameter in its own way. Generally, stream sockets may successfully **connect( )** only once; datagram sockets may use **connect( )** multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

RETURN VALUES

connect( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

The call fails if:

EADDRINUSE	The address is already in use.
EADDRNOTAVAIL	The specified address is not available on the remote machine.
EAFNOSUPPORT	Addresses in the specified address family cannot be used with this socket.
EALREADY	The socket is non-blocking and a previous connection attempt has not yet been completed.
EBADF	*s* is not a valid descriptor.
ECONNREFUSED	The attempt to connect was forcefully rejected. The calling program should **close**(2V) the socket descriptor, and issue another **socket**(2) call to obtain a new descriptor before attempting another **connect**(2) call.
EFAULT	The *name* parameter specifies an area outside the process address space.
EINPROGRESS	The socket is non-blocking and the connection cannot be completed immediately. It is possible to **select**(2) for completion by selecting the socket for writing.
EINTR	The connection attempt was interrupted before any data arrived by the delivery of a signal.
EINVAL	*namelen* is not the size of a valid address for the specified address family.
EISCONN	The socket is already connected.
ENETUNREACH	The network is not reachable from this host.
ENOTSOCK	*s* is a descriptor for a file, not a socket.
ETIMEDOUT	Connection establishment timed out without establishing a connection.

The following errors are specific to connecting names in the UNIX domain. These errors may not apply in future versions of the UNIX IPC domain.

EACCES	Search permission is denied for a component of the path prefix of the path name in *name*.
ELOOP	Too many symbolic links were encountered in translating the path name in *name*.
EIO	An I/O error occurred while reading from or writing to the file system.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} (see **sysconf**(2V)) while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	A component of the path prefix of the path name in *name* does not exist.
	The socket referred to by the path name in *name* does not exist.
ENOTDIR	A component of the path prefix of the path name in *name* is not a directory.
ENOTSOCK	The file referred to by *name* is not a socket.
EPROTOTYPE	The file referred to by *name* is a socket of a type other than the type of *s* (e.g., *s* is a SOCK_DGRAM socket, while *name* refers to a SOCK_STREAM socket).

SEE ALSO

accept(2), **close**(2V), **connect**(2), **getsockname**(2), **select**(2), **socket**(2)

NAME

creat – create a new file

SYNOPSIS

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

SYSTEM V SYNOPSIS

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

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

DESCRIPTION

This interface is made obsolete by **open**(2V), since,

```
creat(path, mode);
```

is equivalent to

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

creat( ) creates a new ordinary file or prepares to rewrite an existing file named by the pathname pointed to by *path*. If the file did not exist, it is given the mode *mode*, as modified by the process's mode mask (see **umask**(2V)). See **stat**(2V) for the construction of *mode*.

If the file exists, its mode and owner remain unchanged, but it is truncated to 0 length. Otherwise, the file's owner ID is set to the effective user ID of the process, and upon successful completion, **creat**( ) marks for update the **st_atime**, **st_ctime**, and **st_mtime** fields of the file (see **stat**(2V)) and the **st_ctime** and **st_mtime** fields of the parent directory.

The file's group ID is set to either:

- the effective group ID of the process, if the filesystem was not mounted with the BSD file-creation semantics flag (see **mount**(2V)) and the set-gid bit of the parent directory is clear, or
- the group ID of the directory in which the file is created.

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**(2V).
- The "save text image after execution" (sticky) bit of the mode is cleared. See **chmod**(2V).
- The "set group ID on execution" bit of the mode is cleared if the effective user ID of the process is not super-user and the process is not a member of the group of the created file.

Upon successful completion, the file descriptor is returned and the file is open for writing, even if the access permissions of the file mode do not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across **execve**(2V) system calls. See **fcntl**(2V).

If the file did not previously exist, upon successful completion, **creat**( ) marks for update the **st_ctime** and **st_mtime** fields of the file and the **st_ctime** and **st_mtime** fields of the parent directory.

RETURN VALUES

creat( ) returns a non-negative descriptor that only permits writing on success. On failure, it returns –1 and sets **errno** to indicate the error.

ERRORS

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

The file referred to by *path* does not exist and the directory in which it is to be created is not writable.

The file referred to by *path* exists, but it is unwritable.

EDQUOT	The directory in which the entry for the new file is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted. The user's quota of inodes on the file system on which the file is being created has been exhausted.
EFAULT	*path* points outside the process's allocated address space.
EINTR	The **creat**( ) operation was interrupted by a signal.
EIO	An I/O error occurred while making the directory entry or allocating the inode.
EISDIR	The file referred to by *path* is a directory.
ELOOP	Too many symbolic links were encountered in translating the pathname pointed to by *path*.
EMFILE	There are already too many files open.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}. A pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENFILE	The system file table is full.
ENOENT	A component of the path prefix does not exist.
ENOSPC	The directory in which the entry for the new file is being placed cannot be extended because there is no space left on the file system containing the directory. There are no free inodes on the file system on which the file is being created.
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The file is a character special or block special file, and the associated device does not exist.
EOPNOTSUPP	The file was a socket (not currently implemented).
EROFS	The file referred to by *path* resides, or would reside, on a read-only file system.

SYSTEM V ERRORS

In addition to the above, the following may also occur:

ENOENT	*path* points to an empty string.

SEE ALSO

close(2V), **chmod**(2V), **execve**(2V), **fcntl**(2V), **flock**(2), **mount**(2V), **open**(2V), **write**(2V), **umask**(2V)

NOTES

The *mode* given is arbitrary; it need not allow writing. This feature has been used in the past by programs to construct a simple exclusive locking mechanism. It is replaced by the O_EXCL open mode, or **flock**(2) facility.

NAME

dup, dup2 – duplicate a descriptor

SYNOPSIS

int dup(fd)
int fd;

int dup2(fd1, fd2)
int fd1, fd2;

DESCRIPTION

dup() duplicates an existing object descriptor. The argument *fd* is a small non-negative integer index in the per-process descriptor table. The value must be less than the size of the table, which is returned by **getdtablesize**(2). The new descriptor returned by the call is the lowest numbered descriptor that is not currently in use by the process.

With **dup2()**, *fd2* specifies the desired value of the new descriptor. If descriptor *fd2* is already in use, it is first deallocated as if it were closed by **close**(2V).

The new descriptor has the following in common with the original:

- It refers to the same object that the old descriptor referred to.
- It uses the same seek pointer as the old descriptor. (that is, both file descriptors share one seek pointer).
- It has the same access mode (read, write or read/write) as the old descriptor.

Thus if *fd2* and *fd1* are duplicate references to an open file, **read**(2V), **write**(2V), and **lseek**(2V) calls all move a single seek pointer into the file, and append mode, non-blocking I/O and asynchronous I/O options are shared between the references. If a separate seek pointer into the file is desired, a different object reference to the file must be obtained by issuing an additional **open**(2V) call. The close-on-exec flag on the new file descriptor is unset.

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

RETURN VALUES

dup() and **dup2()** return a new descriptor on success. On failure, they return –1 and set **errno** to indicate the error.

ERRORS

EBADF	*fd1* or *fd2* is not a valid active descriptor.
EMFILE	Too many descriptors are active.

SEE ALSO

accept(2), **close**(2V), **fcntl**(2V), **getdtablesize**(2), **lseek**(2V), **open**(2V), **pipe**(2V), **read**(2V), **socket**(2), **socketpair**(2), **write**(2V)

NAME

execve – execute a file

SYNOPSIS

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

DESCRIPTION

execve( ) transforms the calling process into a new process. The new process is constructed from an ordinary file, whose name is pointed to by *path*, called the *new process file*. This file is either an executable object file, or a file of data for an interpreter. An executable object file consists of an identifying header, followed by pages of data representing the initial program (text) and initialized data pages. Additional pages may be specified by the header to be initialized with zero data. See **a.out**(5).

An interpreter file begins with a line of the form '**#!** *interpreter* [*arg*]'. Only the first thirty-two characters of this line are significant. When *path* refers to an interpreter file, **execve( )** invokes the specified *interpreter*. If the optional *arg* is specified, it becomes the first argument to the *interpreter*, and the pathname to which *path* points becomes the second argument. Otherwise, the pathname to which *path* points becomes the first argument. The original arguments are shifted over to become the subsequent arguments. The zeroth argument, normally the pathname to which *path* points, is left unchanged.

There can be no return from a successful **execve( )** because the calling process image is lost. This is the mechanism whereby different process images become active.

The argument *argv* is a pointer to a null-terminated array of character pointers to null-terminated character strings. These strings constitute the argument list to be made available to the new process. By convention, at least one argument must be present in this array, and the first element of this array should be the name of the executed program (that is, the last component of *path*).

The argument *envp* is also a pointer to a null-terminated array of character pointers to null-terminated strings. These strings pass information to the new process which are not directly arguments to the command (see **environ**(5V)).

The number of bytes available for the new process's combined argument and environment lists (including null terminators, pointers and alignment bytes) is {ARG_MAX} (see **sysconf**(2V)). On SunOS systems, {ARG_MAX} is currently one megabyte.

Descriptors open in the calling process remain open in the new process, except for those for which the close-on-exec flag is set (see **close**(2V) and **fcntl**(2V)). Descriptors which remain open are unaffected by **execve( )**.

Signals set to the default action (SIG_DFL) in the calling process image are set to the default action in the new process image. Signals set to be ignored (SIG_IGN) by the calling process image are ignored by the new process image. Signals set to be caught by the calling process image are reset to the default action in the new process image. Signals set to be blocked in the calling process image remain blocked in the new process image, regardless of changes to the signal action. The signal stack is reset to be undefined (see **sigvec**(2) for more information).

Each process has a *real* user ID and group ID and an *effective* user ID and group ID. The *real* ID identifies the person using the system; the *effective* ID determines their access privileges. **execve( )** changes the effective user or group ID to the owner or group of the executed file if the file has the "set-user-ID" or "set-group-ID" modes. The *real* UID and GID are not affected. The effective user ID and effective group ID of the new process image are saved as the saved set-user-ID and saved set-group-ID respectively, for use by **setuid**(3V).

execve( ) sets the SEXECED flag for the new process image (see **setpgid**(2V)).

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).

Upon successful completion, **execve**( ) marks for update the **st_atime** field of the file. **execve**( ) also marks **st_atime** for update if it fails, but is able find the process image file.

If **execve**( ) succeeds, the process image file is considered to have been opened (see **open**(2V)). The corresponding close (see **close**(2V)) is considered to occur after the open, but before process termination or successful completion of a subsequent call to **execve**( ).

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

attribute	*see*
process ID	**getpid**(2)
parent process ID	**getpid**(2)
process group ID	**getpgrp**(2V), **setpgid**(2V)
session membership	**setsid**(2)
real user ID	**getuid**(2)
real group ID	**getgid**(2)
supplementary group IDs	**Intro**(2)
time left until an alarm	**alarm**(3C)
supplementary group IDs	**getgroups**(2)
semadj values	**semop**(2)
working directory	**chdir**(2)
root directory	**chroot**(2)
controlling terminal	**termio**(4)
trace flag	**ptrace**(2), request 0
resource usages	**getrusage**(2)
interval timers	**getitimer**(2)
resource limits	**getrlimit**(2)
file mode mask	**umask**(2)
process signal mask	**sigvec**(2), **sigprocmask**(2V), **sigsetmask**(2)
pending signals	**sigpending**(2)
tms_utime, **tms_stime**, **tms_cutime**, **tms_cstime**	**times**(3C)

When the executed program begins, it is called as follows:

```
main(argc, argv, envp)
int argc;
char *argv[ ], *envp[ ];
```

where *argc* is the number of elements in *argv* (the "arg count", not counting the NULL terminating pointer) and *argv* points to the array of character pointers to the arguments themselves.

envp is a pointer to an array of strings that constitute the *environment* of the process. A pointer to this array is also stored in the global variable **environ**. Each string consists of a name, an "=", and a null-terminated value. The array of pointers is terminated by a NULL pointer. The shell **sh**(1) passes an environment entry for each global shell variable defined when the program is called. See **environ**(5V) for some conventionally used names.

Note: Passing values for *argc*, *argv*, and *envp* to **main**( ) is optional.

RETURN VALUES

execve( ) returns to the calling process only on failure. It returns −1 and sets **errno** to indicate the error.

ERRORS

E2BIG	The total number of bytes in the new process file's argument and environment lists exceeds {ARG_MAX} (see **sysconf**(2V)).
EACCES	Search permission is denied for a component of the new process file's path prefix.

	The new process file is not an regular file.
	Execute permission is denied for the new process file.
EFAULT	The new process file is not as long as indicated by the size values in its header.
	path, *argv*, or *envp* points to an illegal address.
EIO	An I/O error occurred while reading from the file system.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} (see **sysconf**(2V)) while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ELOOP	Too many symbolic links were encountered in translating *path*.
ENOENT	One or more components of the path prefix of the new process file does not exist.
	The new process file does not exist.
ENOEXEC	The new process file has the appropriate access permission, but has an invalid magic number in its header.
ENOMEM	The new process file requires more virtual memory than is allowed by the imposed maximum (**getrlimit**(2)).
ENOTDIR	A component of the path prefix of the new process file is not a directory.

SYSTEM V ERRORS

In addition to the above, the following may also occur:

ENOENT	*path* points to a null pathname.

SEE ALSO

sh(1), **chdir**(2V), **chroot**(2), **close**(2V), **exit**(2V), **fcntl**(2V), **fork**(2V), **getgroups**(2V), **getitimer**(2), **getpid**(2V), **getrlimit**(2), **getrusage**(2), **profil**(2), **ptrace**(2), **semop**(2), **getpgrp**(2V), **shmop**(2), **sigvec**(2), **execl**(3V), **setuid**(3V), **termio**(4), **a.out**(5), **environ**(5V)

WARNINGS

If a program is **setuid()** to a non-super-user, but is executed when the real user ID is super-user, then the program has some of the powers of a super-user as well.

NAME

_exit – terminate a process

SYNOPSIS

void _exit(status)
int status;

DESCRIPTION

_exit( ) terminates a process with the following consequences:

All of the descriptors open in the calling process are closed. This may entail delays, for example, waiting for output to drain; a process in this state may not be killed, as it is already dying.

If the parent process of the calling process is executing a **wait( )** or **waitpid( )**, or is interested in the **SIGCHLD** signal, then it is notified of the calling process's termination and the low-order eight bits of *status* are made available to it (see **wait**(2V)).

If the parent process of the calling process is not executing a **wait( )** or **waitpid( )**, *status* is saved for return to the parent process whenever the parent process executes an appropriate subsequent **wait( )** or **waitpid( )**.

The parent process ID of all of the calling process's existing child processes are also set to 1. This means that the initialization process (see **intro**(2)) inherits each of these processes as well. Any stopped children are restarted with a hangup signal (**SIGHUP**).

If the process is a controlling process, **SIGHUP** is sent to each process in the foreground process group of the controlling terminal belonging to the calling process, and the controlling terminal associated with the session is disassociated from the session, allowing it to be acquired by a new controlling process (see **setsid**(2V)).

If **_exit( )** causes a process group to become orphaned, and if any member of the newly-orphaned process group is stopped, then **SIGHUP** followed by **SIGCONT** is sent to each process in the newly-orphaned process group (see **setpgid**(2V)).

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 process accounting is enabled (see **acct**(2V)), an accounting record is written to the accounting file.

Most C programs will call the library routine **exit**(3) which performs cleanup actions in the standard I/O library before calling **_exit( )**.

RETURN VALUES

_exit( ) never returns.

SEE ALSO

intro(2), **acct**(2V), **fork**(2V), **semop**(2), **wait**(2V), **exit**(3)

NAME

fcntl – file control

SYNOPSIS

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

int fcntl(fd, cmd, arg)
int fd, cmd, arg;
```

DESCRIPTION

fcntl( ) performs a variety of functions on open descriptors. The argument *fd* is an open descriptor used by *cmd* as follows:

F_DUPFD Returns a new descriptor, which has the smallest value greater than or equal to *arg*. It refers to the same object as the original descriptor, and has the same access mode (read, write or read/write). The new descriptor shares descriptor status flags with *fd*, and if the object was a file, the same file pointer. It is also associated with a **FD_CLOEXEC** (close-on-exec) flag set to remain open across **execve**(2V) system calls.

F_GETFD Get the **FD_CLOEXEC** (close-on-exec) flag associated with *fd*. If the low-order bit is **0**, the file remains open after executing **execve( )**, otherwise it is closed.

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

Note: this is a per-process and per-descriptor flag. Setting or clearing it for a particular descriptor does not affect the flag on descriptors copied from it by **dup**(2V) or **F_DUPFD**, nor does it affect the flag on other processes of that descriptor.

F_GETFL Get descriptor status flags (see **fcntl**(5) for definitions).

F_SETFL Set descriptor status flags (see **fcntl**(5) for definitions). The following flags are the only ones whose values may change: **O_APPEND**, **O_SYNC**, and **O_NDELAY**, and the **FASYNC**, **FNDELAY**, and **FNBIO** flags defined in **<fcntl.h>**.

O_NDELAY and **FNDELAY** are identical.

Descriptor status flag values set by **F_SETFL** affects descriptors copied using **dup**(2V), **F_DUPFD** or other processes.

Setting or clearing the **FNDELAY** flag on a descriptor causes an **FIONBIO** **ioctl**(2) request to be performed on the object referred to by that descriptor. Setting or clearing non-blocking mode, and setting or clearing the **FASYNC** flag on a descriptor causes an **FIOASYNC** **ioctl**(2) request to be performed on the object referred to by that descriptor, setting or clearing asynchronous mode. Thus, all descriptors referring to the object are affected.

F_GETLK Get a description of the first lock which would block the lock specified by the **flock** structure pointed to by *arg* (see the definition of **struct flock** below). If a lock exists, The **flock** structure is overwritten with that lock's description. Otherwise, the structure is passed back with the lock type set to **F_UNLOCK** and is otherwise unchanged.

F_SETLK Set or clear a file segment lock according to the **flock** structure pointed to by *arg*. **F_SETLK** is used to set shared (**F_RDLCK**) or exclusive (**F_WRLCK**) locks, or to remove those locks (**F_UNLCK**). If the specified lock cannot be applied, **fcntl( )** fails and returns immediately.

F_SETLKW This *cmd* is the same as **F_SETLK** except that if a shared or exclusive lock is blocked by other locks, the process waits until the requested lock can be applied. If a signal that is set to be caught (see **signal**(3V)) is received while **fcntl()** is waiting for a region, the call to **fcntl()** is interrupted. Upon return from the process's signal handler, **fcntl()** fails and returns, and the requested lock is not applied.

F_GETOWN Get the process ID or process group currently receiving **SIGIO** and **SIGURG** signals; process groups are returned as negative values.

F_SETOWN Set the process or process group to receive **SIGIO** and **SIGURG** signals. Process groups are specified by supplying *arg* as negative, otherwise *arg* is interpreted as a process ID.

F_RSETLK
F_RSETLKW
F_RGETLK Are used by the network lock daemon, **lockd**(8C), to communicate with the NFS server kernel to handle locks on the NFS files.

Record locking is done with either *shared* (**F_RDLCK**), or *exclusive* (**F_WRLCK**) locks. More than one process may hold a shared lock on a particular file segment, but if one process holds an exclusive lock on the segment, no other process may hold any lock on the segment until the exclusive lock is removed.

In order to claim a shared lock, a descriptor must be opened with read access. Descriptors for exclusive locks must be opened with write access.

A shared lock may be changed to an exclusive lock, and vice versa, simply by specifying the appropriate lock type with a **F_SETLK** or **F_SETLKW** *cmd*. Before the previous lock is released and the new lock applied, any other processes already in line must gain and release their locks.

If *cmd* is **F_SETLKW** and the requested lock cannot be claimed immediately (for instance, when another process holds an exclusive lock that overlaps the current request) the calling process is blocked until the lock may be acquired. These blocks may be interrupted by signals. Care should be taken to avoid deadlocks caused by multiple processes all blocking the same records.

A shared or exclusive lock is either *advisory* or *mandatory* depending on the mode bits of the file containing the locked segment. The lock is mandatory if the set-GID bit (**S_ISGID**) is set and the group execute bit (**S_IXGRP**) is clear (see **stat**(2V) for information about mode bits). Otherwise, the lock is advisory.

If a process holds a mandatory shared lock on a segment of a file, other processes may read from the segment, but write operations block until all locks are removed. If a process holds a mandatory exclusive lock on a segment of a file, both read and write operations block until the lock is removed (see WARNINGS).

An advisory lock does not affect read and write access to the locked segment. Advisory locks may be used by cooperating processes checking for locks using **F_GETLCK** and voluntarily observing the indicated read and write restrictions.

The record to be locked or unlocked is described by the **flock** structure defined in **<fcntl.h>** as follows:

```
struct flock {
        short l_type;     /* F_RDLCK, F_WRLCK, or F_UNLCK */
        short l_whence;   /* flag to choose starting offset */
        long  l_start;    /* relative offset, in bytes */
        long  l_len;      /* length, in bytes; 0 means lock to EOF */
        pid_t l_pid;      /* returned with F_GETLK */
};
```

The **flock** structure describes the type (**l_type**), starting offset (**l_whence**), relative offset (**l_start**), and size (**l_len**) of the file segment to be affected. **l_whence** is set to **SEEK_SET**, **SEEK_CUR**, or **SEEK_END** (see **lseek**(2V)) to indicate that the relative offset is to be measured from the start of the file, current position, or EOF, respectively. The process id field (**l_pid**) is only used with the **F_GETLK** *cmd* to return the description of a lock held by another process. Note: do not confuse **struct flock** with the function **flock**(2). They are unrelated.

Locks may start or extend beyond the current EOF, but may not be negative relative to the beginning of the file. Setting **l_len** to zero (0) extends the lock to EOF. If **l_whence** is set to **SEEK_SET** and **l_start** and **l_len** are set to zero (0), the entire file is locked. Changing or unlocking the subset of a locked segment leaves the smaller segments at either end locked. Locking a segment already locked by the calling process causes the old lock type to be removed and the new lock type to take affect. All locks associated with a file for a given process are removed when the file is closed or the process terminates. Locks are not inherited by the child process in a **fork**(2V) system call.

fcntl( ) record locks are implemented in the kernel for local locks, and throughout the network by the network lock daemon (**lockd**(8C)) for remote locks on NFS files. If the file server crashes and has to be rebooted, the lock daemon attempts to recover all locks that were associated with that server. If a lock cannot be reclaimed, the process that held the lock is issued a **SIGLOST** signal.

In order to maintain consistency in the network case, data must not be cached on client machines. For this reason, file buffering for an NFS file is turned off when the first lock is attempted on the file. Buffering remains off as long as the file is open. Programs that do I/O buffering in the user address space, however, may have inconsistent results. The standard I/O package, for instance, is a common source of unexpected buffering.

SYSTEM V DESCRIPTION

O_NDELAY and **FNBIO** are identical.

RETURN VALUES

On success, the value returned by **fcntl**( ) depends on *cmd* as follows:

F_DUPFD	A new descriptor.
F_GETFD	Value of flag (only the low-order bit is defined).
F_GETFL	Value of flags.
F_GETOWN	Value of descriptor owner.
other	Value other than −1.

On failure, **fcntl**( ) returns −1 and sets **errno** to indicate the error.

ERRORS

EACCES — *cmd* is **F_SETLK**, the lock type (**l_type**) is **F_RDLCK** (shared lock), and the file segment to be locked is already under an exclusive lock held by another process. This error is also returned if the lock type is **F_WRLCK** (exclusive lock) and the file segment is already locked with a shared or exclusive lock.

Note: In future, **fcntl**( ) may generate EAGAIN under these conditions, so applications testing for EACCES should also test for EAGAIN.

EBADF — *fd* is not a valid open descriptor.

cmd is **F_SETLK** or **F_SETLKW** and the process does not have the appropriate read or write permissions on the file.

EDEADLK — *cmd* is **F_SETLKW**, the lock is blocked by one from another process, and putting the calling-process to sleep would cause a deadlock.

EFAULT — *cmd* is **F_GETLK**, **F_SETLK**, or **F_SETLKW** and *arg* points to an invalid address.

EINTR — *cmd* is **F_SETLKW** and a signal interrupted the process while it was waiting for the lock to be granted.

EINVAL — *cmd* is **F_DUPFD** and *arg* is negative or greater than the maximum allowable number (see **getdtablesize**(2)).

cmd is **F_GETLK**, **F_SETLK**, or **F_SETLKW** and *arg* points to invalid data.

EMFILE *cmd* is **F_DUPFD** and the maximum number of open descriptors has been reached.

ENOLCK *cmd* is **F_SETLK** or **F_SETLKW** and there are no more file lock entries available.

SEE ALSO

close(2V), **execve**(2V), **flock**(2), **fork**(2V), **getdtablesize**(2), **ioctl**(2), **open**(2V), **sigvec**(2), **lockf**(3), **fcntl**(5), **lockd**(8C)

WARNINGS

Mandatory record locks are dangerous. If a runaway or otherwise out-of-control process should hold a mandatory lock on a file critical to the system and fail to release that lock, the entire system could hang or crash. For this reason, mandatory record locks may be removed in a future SunOS release. Use advisory record locking whenever possible.

NOTES

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee exclusive access. Files can be accessed without advisory files, but inconsistencies may result.

read(2V) and **write**(2V) system calls on files are affected by mandatory file and record locks (see **chmod**(2V)).

BUGS

File locks obtained by **fcntl**() do not interact with **flock**() locks. They do, however, work correctly with the exclusive locks claimed by **lockf**(3).

F_GETLK returns **F_UNLCK** if the requesting process holds the specified lock. Thus, there is no way for a process to determine if it is still holding a specific lock after catching a **SIGLOST** signal.

In a network environment, the value of **l_pid** returned by **F_GETLK** is next to useless.

NAME

flock – apply or remove an advisory lock on an open file

SYNOPSIS

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

#define LOCK_SH     1     /* shared lock */
#define LOCK_EX     2     /* exclusive lock */
#define LOCK_NB     4     /* don't block when locking */
#define LOCK_UN     8     /* unlock */

int flock(fd, operation)
int fd, operation;
```

DESCRIPTION

flock() applies or removes an *advisory* lock on the file associated with the file descriptor *fd*. A lock is applied by specifying an *operation* parameter that is the inclusive OR of LOCK_SH or LOCK_EX and, possibly, LOCK_NB. To unlock an existing lock, the *operation* should be LOCK_UN.

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee exclusive access (that is, processes may still access files without using advisory locks, possibly resulting in inconsistencies).

The locking mechanism allows two types of locks: *shared* locks and *exclusive* locks. More than one process may hold a shared lock for a file at any given time, but multiple exclusive, or both shared and exclusive, locks may not exist simultaneously on a file.

A shared lock may be *upgraded* to an exclusive lock, and vice versa, simply by specifying the appropriate lock type; the previous lock will be released and the new lock applied (possibly after other processes have gained and released the lock).

Requesting a lock on an object that is already locked normally causes the caller to block until the lock may be acquired. If LOCK_NB is included in *operation*, then this will not happen; instead the call will fail and the error EWOULDBLOCK will be returned.

NOTES

Locks are on files, not file descriptors. That is, file descriptors duplicated through **dup**(2V) or **fork**(2V)) do not result in multiple instances of a lock, but rather multiple references to a single lock. If a process holding a lock on a file forks and the child explicitly unlocks the file, the parent will lose its lock.

Processes blocked awaiting a lock may be awakened by signals.

RETURN VALUES

flock() returns:

0	on success.
−1	on failure and sets **errno** to indicate the error.

ERRORS

EBADF	The argument **fd** is an invalid descriptor.
EOPNOTSUPP	The argument **fd** refers to an object other than a file.
EWOULDBLOCK	The file is locked and the LOCK_NB option was specified.

SEE ALSO

close(2V), **dup**(2V), **execve**(2V), **fcntl**(2V), **fork**(2V), **open**(2V), **lockf**(3), **lockd**(8C)

BUGS

Locks obtained through the **flock()** mechanism are known only within the system on which they were placed. Thus, multiple clients may successfully acquire exclusive locks on the same remote file. If this behavior is not explicitly desired, the **fcntl**(2V) or **lockf**(3) system calls should be used instead; these make use of the services of the **network lock manager** (see **lockd**(8C)).

NAME

fork – create a new process

SYNOPSIS

int fork()

SYSTEM V SYNOPSIS

pid_t fork()

DESCRIPTION

fork() creates a new process. The new process (child process) is an exact copy of the calling process except for the following:

- The child process has a unique process ID. The child process ID also does not match any active process group ID.
- The child process has a different parent process ID (the process ID of the parent process).
- The child process has its own copy of the parent's descriptors. These descriptors reference the same underlying objects, so that, for instance, file pointers in file objects are shared between the child and the parent, so that an **lseek**(2V)) on a descriptor in the child process can affect a subsequent **read**(2V) or **write**(2V) by the parent. This descriptor copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.
- The child process has its own copy of the parent's open directory streams (see **directory**(3V)). Each open directory stream in the child process shares directory stream positioning with the corresponding directory stream of the parent.
- All *semadj* values are cleared; see **semop**(2).
- The child processes resource utilizations are set to 0; see **getrlimit**(2). The **it_value** and **it_interval** values for the **ITIMER_REAL** timer are reset to 0; see **getitimer**(2).
- The child process's values of **tms_utime()**, **tms_stime()**, **tms_cutime()**, and **tms_cstime()** (see **times**(3V)) are set to zero.
- File locks (see **fcntl**(2V)) previously set by the parent are not inherited by the child.
- Pending alarms (see **alarm**(3V)) are cleared for the child process.
- The set of signals pending for the child process is cleared (see **sigvec**(2)).

RETURN VALUES

On success, **fork()** returns 0 to the child process and returns the process ID of the child process to the parent process. On failure, **fork()** returns –1 to the parent process, sets **errno** to indicate the error, and no child process is created.

ERRORS

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. This limit is determined when the system is generated.

The system-imposed limit on the total number of processes under execution by a single user would be exceeded. This limit is determined when the system is generated.

ENOMEM — There is insufficient swap space for the new process.

SEE ALSO

execve(2V), **getitimer**(2), **getrlimit**(2), **lseek**(2V), **read**(2V), **semop**(2), **wait**(2V), **write**(2V)

NAME

fsync – synchronize a file's in-core state with that on disk

SYNOPSIS

int fsync(fd)

int fd;

DESCRIPTION

fsync( ) moves all modified data and attributes of *fd* to a permanent storage device: all in-core modified copies of buffers for the associated file have been written to a disk when the call returns. Note: this is different than **sync**(2) which schedules disk I/O for all files (as though an **fsync( )** had been done on all files) but returns before the I/O completes.

fsync( ) should be used by programs which require a file to be in a known state; for example, a program which contains a simple transaction facility might use it to ensure that all modifications to a file or files caused by a transaction were recorded on disk.

RETURN VALUES

fsync( ) returns:

0	on success.
–1	on failure and sets **errno** to indicate the error.

ERRORS

EBADF	*fd* is not a valid descriptor.
EINVAL	*fd* refers to a socket, not a file.
EIO	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

cron(8), **sync**(2)

NAME

getauid, setauid – get and set user audit identity

SYNOPSIS

int getauid()

int setauid(auid)
int auid;

DESCRIPTION

The **getauid()** system call returns the audit user ID for the current process. This value is initially set at login time and inherited by all child processes. This value does not change when the real/effective user IDs change, so it can be used to identify the logged-in user, even when running a setuid program. The audit user ID governs audit decisions for a process.

The **setauid()** system call sets the audit user ID for the current process. Only the super-user may successfully execute these calls.

RETURN VALUES

getauid() returns the audit user ID of the current process on success. On failure, it returns –1 and sets **errno** to indicate the error.

setauid() returns:

0 on success.

–1 on failure and sets **errno** to indicate the error.

ERRORS

EINVAL The parameter *auid* is not a valid UID.

EPERM The process's effective user ID is not super-user.

SEE ALSO

getuid(2V), **setuseraudit**(2), **audit**(8)

NAME

getdents – gets directory entries in a filesystem independent format

SYNOPSIS

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

int getdents(fd, buf, nbytes)
int fd;
char *buf;
int nbytes;
```

DESCRIPTION

getdents( ) attempts to put directory entries from the directory referenced by the file descriptor *fd* into the buffer pointed to by *buf*, in a filesystem independent format. Up to *nbytes* bytes of data will be transferred.

The data in the buffer is a series of **dirent** structures each containing the following entries:

```
off_t       d_off;
u_long      d_fileno;
u_short     d_reclen;
u_short     d_namlen;
char        d_name[MAXNAMLEN + 1];  /* see below */
```

The **d_off** entry contains a value which is interpretable only by the filesystem that generated it. It may be supplied as an offset to **lseek**(2V)) to find the entry following the current one in a directory. The **d_fileno** entry is a number which is unique for each distinct file in the filesystem. Files that are linked by hard links (see **link**(2V)) have the same **d_fileno**. The **d_reclen** entry is the length, in bytes, of the directory record. The **d_name** entry contains a null terminated file name. The **d_namlen** entry specifies the length of the file name. Thus the actual size of **d_name** may vary from 1 to MAXNAMLEN+1.

The structures are not necessarily tightly packed. The **d_reclen** entry may be used as an offset from the beginning of a *dirent* structure to the next structure, if any.

Upon return, the actual number of bytes transferred is returned. The current position pointer associated with *fd* is set to point to the directory entry following the last one returned. The pointer is not necessarily incremented by the number of bytes returned by **getdents( )**. If the value returned is zero, the end of the directory has been reached. The current position pointer may be set and retrieved by **lseek**(2V). It is not safe to set the current position pointer to any value other than a value previously returned by **lseek**(2V), or the value of a **d_off** entry in a **dirent** structure returned by **getdents( )**, or zero.

RETURN VALUES

getdents( ) returns the number of bytes actually transferred on success. On failure, it returns −1 and sets **errno** to indicate the error.

ERRORS

EBADF	*fd* is not a valid file descriptor open for reading.
EFAULT	*buf* points outside the allocated address space.
EINTR	A read from a slow device was interrupted before any data arrived by the delivery of a signal.
EINVAL	*nbytes* is not large enough for one directory entry.
ENOTDIR	The file referenced by *fd* is not a directory.
EIO	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

link(2V), **lseek**(2V), **open**(2V), **directory**(3V)

NOTES

It is strongly recommended, for portability reasons, that programs that deal with directory entries use the **directory**(3V) interface rather than directly calling **getdents**().

NAME

getdirentries – gets directory entries in a filesystem independent format

SYNOPSIS

```
int getdirentries(fd, buf, nbytes, basep)
int fd;
char *buf;
int nbytes;
long *basep;
```

DESCRIPTION

This system call is now obsolete. It is superseded by the **getdents**(2) system call, which returns directory entries in a new format specified in **<sys/dirent.h>**. The file, **<sys/dir.h>**, has also been modified to use the new directory entry format. Programs which currently call **getdirentries( )** should be modified to use the new system call and the new include file **dirent.h** or, preferably, to use the **directory**(3V) library routines. The **getdirentries( )** system call is retained in the current SunOS release only for purposes of backwards binary compatibility and will be removed in a future major release.

getdirentries( ) attempts to put directory entries from the directory referenced by the file descriptor **fd** into the buffer pointed to by *buf*, in a filesystem independent format. Up to *nbytes* bytes of data will be transferred. *nbytes* must be greater than or equal to the block size associated with the file, see **stat**(2V). Sizes less than this may cause errors on certain filesystems.

The data in the buffer is a series of structures each containing the following entries:

```
unsigned long  d_fileno;
unsigned short d_reclen;
unsigned short d_namlen;
char           d_name[MAXNAMELEN + 1];  /* see below */
```

The **d_fileno** entry is a number which is unique for each distinct file in the filesystem. Files that are linked by hard links (see **link**(2V)) have the same **d_fileno**. The **d_reclen** entry is the length, in bytes, of the directory record. The **d_name** entry contains a null terminated file name. The **d_namlen** entry specifies the length of the file name. Thus the actual size of **d_name** may vary from 2 to MAXNAMELEN+1.

The structures are not necessarily tightly packed. The **d_reclen** entry may be used as an offset from the beginning of a **direct** structure to the next structure, if any.

Upon return, the actual number of bytes transferred is returned. The current position pointer associated with **fd** is set to point to the next block of entries. The pointer is not necessarily incremented by the number of bytes returned by **getdirentries( )**. If the value returned is zero, the end of the directory has been reached. The current position pointer may be set and retrieved by **lseek**(2V). **getdirentries( )** writes the position of the block read into the location pointed to by *basep*. It is not safe to set the current position pointer to any value other than a value previously returned by **lseek**(2V) or a value previously returned in the location pointed to by *basep* or zero.

RETURN VALUES

getdirentries( ) returns the number of bytes actually transferred on success. On failure, it returns –1 and sets **errno** to indicate the error.

ERRORS

EBADF	*fd* is not a valid file descriptor open for reading.
EFAULT	Either *buf* or *basep* points outside the allocated address space.
EINTR	A read from a slow device was interrupted before any data arrived by the delivery of a signal.
EIO	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

getdents(2), **link**(2V), **lseek**(2V), **open**(2V), **stat**(2V), **directory**(3V)

NAME

getdomainname, setdomainname – get/set name of current domain

SYNOPSIS

```
int getdomainname(name, namelen)
char *name;
int namelen;

int setdomainname(name, namelen)
char *name;
int namelen;
```

DESCRIPTION

getdomainname( ) returns the name of the domain for the current processor, as previously set by **setdomainname**. The parameter *namelen* specifies the size of the array pointed to by *name*. The returned name is null-terminated unless insufficient space is provided.

setdomainname( ) sets the domain of the host machine to be *name*, which has length *namelen*. This call is restricted to the super-user and is normally used only when the system is bootstrapped.

The purpose of domains is to enable two distinct networks that may have host names in common to merge. Each network would be distinguished by having a different domain name. At the current time, only the Network Information Service (NIS) and **sendmail**(8) make use of domains.

RETURN VALUES

getdomainname( ) and **setdomainname**( ) return:

0 on success.

−1 on failure and set **errno** to indicate the error.

ERRORS

EFAULT The *name* parameter gave an invalid address.

In addition to the above, **setdomainname**( ) will fail if:

EPERM The caller was not the super-user.

NOTES

Domain names are limited to 64 characters.

The Network Information Service (NIS) was formerly known as Sun Yellow Pages (YP). The functionality of the two remains the same; only the name has changed.

NAME

getdtablesize – get descriptor table size

SYNOPSIS

getdtablesize( )

DESCRIPTION

The call **getdtablesize( )** returns the current value of the soft limit component of the **RLIMIT_NOFILE** resource limit. This resource limit governs the maximum value allowable as the index of a newly created descriptor.

WARNINGS

getdtablesize is implemented as a system call only for binary compatibility with previous releases.

Because of possible intervening **getrlimit**(2) calls affecting **RLIMIT_NOFILE**, repeated calls to **getdtablesize( )** may return different values. Thus it is unwise to cache the return value in an effort to avoid system call overhead, unless it is known that such intervening calls do not occur.

SEE ALSO

close(2V), **dup**(2V), **getrlimit**(2), **open**(2V)

NAME

getgid, getegid – get group identity

SYNOPSIS

int getgid()

int getegid()

SYSTEM V SYNOPSIS

#include <sys/types.h>

gid_t getgid()

gid_t getegid()

DESCRIPTION

getgid() returns the real group ID of the current process. **getegid()** returns the effective group ID of the current process.

The GID is specified at login time by the group field in the **/etc/passwd** database (see **passwd**(5)).

The effective GID is more transient, and determines additional access permission during execution of a set-GID process, and it is for such processes that **getegid()** is most useful.

SEE ALSO

getuid(2V), **setregid**(2), **setuid**(3V)

NAME

getgroups, setgroups – get or set supplementary group IDs

SYNOPSIS

```
int getgroups(gidsetlen, gidset)
int gidsetlen;
int gidset[];

int setgroups(ngroups, gidset)
int ngroups;
int gidset[];
```

SYSTEM V SYNOPSIS

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

int getgroups(gidsetlen, gidset)
int gidsetlen;
gid_t gidset[];

int setgroups(ngroups, gidset)
int ngroups;
gid_t gidset[];
```

DESCRIPTION

getgroups( ) gets the current supplementary group IDs of the user process and stores it in the array *gidset*. The parameter *gidsetlen* indicates the number of entries that may be placed in *gidset*. **getgroups( )** returns the actual number of entries placed in the *gidset* array. No more than {NGROUPS_MAX} (see **sysconf**(2V)), will ever be returned. If *gidsetlen* is 0, **getgroups( )** returns the number of groups without modifying the *gidset* array.

setgroups( ) sets the supplementary group IDs of the current user process according to the array *gidset*. The parameter *ngroups* indicates the number of entries in the array and must be no more than {NGROUPS_MAX} (see **sysconf**(2V)).

Only the super-user may set new groups.

RETURN VALUES

On success, **getgroups( )** returns the number of entries placed in the array pointed to by *gidset*. On failure, it returns –1 and sets **errno** to indicate the error.

setgroups( ) returns:

0 on success.

–1 on failure and sets **errno** to indicate the error.

ERRORS

Either call fails if:

EFAULT The address specified for *gidset* is outside the process address space.

getgroups( ) fails if:

EINVAL The argument *gidsetlen* is smaller than the number of groups in the group set.

setgroups( ) fails if:

EPERM The caller is not the super-user.

SEE ALSO

initgroups(3)

NAME

gethostid – get unique identifier of current host

SYNOPSIS

gethostid()

DESCRIPTION

gethostid() returns the 32-bit identifier for the current host, which should be unique across all hosts. On a Sun workstation, this number is taken from the CPU board's ID PROM.

SEE ALSO

hostid(1)

NAME

gethostname, sethostname – get/set name of current host

SYNOPSIS

```
int gethostname(name, namelen)
char *name;
int namelen;

int sethostname(name, namelen)
char *name;
int namelen;
```

DESCRIPTION

gethostname( ) returns the standard host name for the current processor, as previously set by **sethostname( )**. The parameter *namelen* specifies the size of the array pointed to by *name*. The returned name is null-terminated unless insufficient space is provided.

sethostname( ) sets the name of the host machine to be *name*, which has length *namelen*. This call is restricted to the super-user and is normally used only when the system is bootstrapped.

RETURN VALUES

gethostname( ) and **sethostname( )** return:

0 on success.

–1 on failure and set **errno** to indicate the error.

ERRORS

EFAULT The *name* or *namelen* parameter gave an invalid address.

In addition to the above, **sethostname( )** may set **errno** to:

EPERM The caller was not the super-user.

SEE ALSO

gethostid(2)

NOTES

Host names are limited to MAXHOSTNAMELEN (from **<sys/param.h>**) characters, currently 64.

NAME

getitimer, setitimer – get/set value of interval timer

SYNOPSIS

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

int getitimer (which, value)
int which;
struct itimerval *value;

int setitimer (which, value, ovalue)
int which;
struct itimerval *value, *ovalue;
```

DESCRIPTION

The system provides each process with three interval timers, defined in **<sys/time.h>**. The **getitimer( )** call stores the current value of the timer specified by **which** into the structure pointed to by *value*. The **setitimer( )** call sets the value of the timer specified by **which** to the value specified in the structure pointed to by *value*, and if *ovalue* is not a NULL pointer, stores the previous value of the timer in the structure pointed to by *ovalue*.

A timer value is defined by the **itimerval** structure, which includes the following members:

```
struct timevalit_interval;/* timer interval */
struct timevalit_value;   /* current value */
```

If **it_value** is non-zero, it indicates the time to the next timer expiration. If **it_interval** is non-zero, it specifies a value to be used in reloading **it_value** when the timer expires. Setting **it_value** to zero disables a timer; however, **it_value** and **it_interval** must still be initialized. Setting **it_interval** to zero causes a timer to be disabled after its next expiration (assuming **it_value** is non-zero).

Time values smaller than the resolution of the system clock are rounded up to this resolution.

The three timers are:

ITIMER_REAL	Decrements in real time. A **SIGALRM** signal is delivered when this timer expires.
ITIMER_VIRTUAL	Decrements in process virtual time. It runs only when the process is executing. A **SIGVTALRM** signal is delivered when it expires.
ITIMER_PROF	Decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by interpreters in statistically profiling the execution of interpreted programs. Each time the **ITIMER_PROF** timer expires, the **SIGPROF** signal is delivered. Because this signal may interrupt in-progress system calls, programs using this timer must be prepared to restart interrupted system calls.

RETURN VALUES

getitimer( ) and **setitimer( )** return:

0	on success.
–1	on failure and set **errno** to indicate the error.

ERRORS

The possible errors are:

EFAULT	The *value* or *ovalue* parameter specified a bad address.
EINVAL	The *value* parameter specified a time that was too large to be handled.

SEE ALSO

sigvec(2), **gettimeofday**(2)

NOTES

Three macros for manipulating time values are defined in **<sys/time.h>**. **timerclear** sets a time value to zero, **timerisset** tests if a time value is non-zero, and **timercmp** compares two time values (beware that >= and <= do not work with this macro).

NAME

getmsg – get next message from 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 that 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 a NULL pointer 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 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 RETURN VALUES. 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 process 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*.

RETURN VALUES

getmsg() returns a non-negative value on success:

0	A full message was read successfully.
MORECTL	More control information is waiting for retrieval. Subsequent **getmsg()** calls will retrieve the rest of the message.
MOREDATA	More data are waiting for retrieval. Subsequent **getmsg()** calls will retrieve the rest of the message.
MORECTL | MOREDATA	Both types of information remain.

On failure, **getmsg()** returns −1 and sets **errno** to indicate the error.

ERRORS

EAGAIN	The O_NDELAY flag is set, and no messages are available.
EBADF	*fd* is not a valid file descriptor open for reading.
EBADMSG	The 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*.
	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), **poll**(2), **putmsg**(2), **read**(2V), **write**(2V)

NAME

getpagesize – get system page size

SYNOPSIS

int getpagesize( )

DESCRIPTION

getpagesize( ) returns the number of bytes in a page. Page granularity is the granularity of many of the memory management calls.

The page size is a *system* page size and may not be the same as the underlying hardware page size.

SEE ALSO

pagesize(1), **brk**(2)

NAME

getpeername – get name of connected peer

SYNOPSIS

```
int getpeername(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

DESCRIPTION

getpeername( ) returns the name of the peer connected to socket *s*. The **int** pointed to by the *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes). The name is truncated if the buffer provided is too small.

DIAGNOSTICS

A 0 is returned if the call succeeds, –1 if it fails.

ERRORS

EBADF	The argument *s* is not a valid descriptor.
EFAULT	The *name* parameter points to memory not in a valid part of the process address space.
ENOBUFS	Insufficient resources were available in the system to perform the operation.
ENOTCONN	The socket is not connected.
ENOTSOCK	The argument *s* is a file, not a socket.

SEE ALSO

accept(2), **bind**(2), **getsockname**(2), **socket**(2)

NAME

getpgrp, setpgrp – return or set the process group of a process

SYNOPSIS

int getpgrp(pid)
int pid;

int setpgrp(pid, pgrp)
int pgrp;
int pid;

SYSTEM V SYNOPSIS

int getpgrp()

int setpgrp()

DESCRIPTION

getpgrp() returns the process group of the process indicated by *pid*. If *pid* is zero, then the call applies to the calling process.

Process groups are used for distribution of signals, and by terminals to arbitrate requests for their input. Processes that have the same process group as the terminal run in the foreground and may read from the terminal, while others block with a signal when they attempt to read.

This call is thus used by programs such as **csh**(1) to create process groups in implementing job control. The **TIOCGPGRP** and **TIOCSPGRP** calls described in **termio**(4) are used to get/set the process group of the control terminal.

setpgrp() sets the process group of the specified process, (*pid*) to the process group specified by *pgrp*. If *pid* is zero, then the call applies to the current (calling) process. If *pgrp* is zero and *pid* refers to the calling process, **setpgrp()** behaves identically to **setsid**(2V).

If the effective user ID of the calling process is not super-user, then the process to be affected must have the same effective user ID as that of the calling process or be a member of the same session as the calling process.

SYSTEM V DESCRIPTION

getpgrp() returns the process group of the calling process.

setpgrp() behaves identically to **setsid()**.

RETURN VALUES

getpgrp() returns the process group of the indicated process on success. On failure, it returns –1 and sets **errno** to indicate the error.

setpgrp() returns:

0 on success.

–1 on failure and sets **errno** to indicate the error.

SYSTEM V RETURN VALUES

getpgrp() returns the process group of the calling process on success.

ERRORS

setpgrp() fails, and the process group is not altered when one of the following occurs:

EACCES The value of *pid* matches the process ID of a child process of the calling process and the child process has successfully executed one of the **exec()** functions.

EINVAL The value of *pgrp* is less than zero or is greater than MAXPID, the maximum process ID as defined in **<sys/param.h>**.

EPERM The process indicated by *pid* is a session leader.

The value of *pid* is valid but matches the process ID of a child process of the calling process and the child process is not in the same session as the calling process.

The value of *pgrp* does not match the process ID of the process indicated by *pid* and there is no process with a process group ID that matches the value of *pgrp* in the same session as the calling process.

The requested process has a different effective user ID from that of the calling process and is not a descendent of the calling process.

The calling process is already a process group leader

The process ID of the calling process equals the process group ID of a different process.

ESRCH The value of *pid* does not match the process ID of the calling process or of a child process of the calling process.

The requested process does not exist.

SEE ALSO

csh(1), **intro**(2), **execve**(2V), **fork**(2V), **getpid**(2V), **getuid**(2V), **kill**(2V), **setpgid**(2V), **signal**(3V), **termio**(4)

NAME

getpid, getppid – get process identification

SYNOPSIS

int getpid()

int getppid()

SYSTEM V SYNOPSIS

#include <sys/types.h>

pid_t getpid()

pid_t getppid()

DESCRIPTION

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

getppid() returns the process ID of the parent of the current process.

SEE ALSO

gethostid(2)

NAME

getpriority, setpriority – get/set process nice value

SYNOPSIS

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority(which, who)
int which, who;

int setpriority(which, who, niceval)
int which, who, niceval;
```

DESCRIPTION

The nice value of a process, process group, or user, as indicated by *which* and *who* is obtained with the **getpriority( )** call and set with the **setpriority( )** call. Process nice values can range from –20 through 19. The default nice value is 0; lower nice values cause more favorable scheduling.

which is one of **PRIO_PROCESS**, **PRIO_PGRP**, or **PRIO_USER**, and *who* is interpreted relative to *which* (a process identifier for **PRIO_PROCESS**, process group identifier for **PRIO_PGRP**, and a user ID for **PRIO_USER**). A zero value of *who* denotes the current process, process group, or user.

The **getpriority( )** call returns the lowest numerical nice value of any of the specified processes. The **setpriority( )** call sets the nice values of all of the specified processes to the value specified by *niceval*. If *niceval* is less than –20, a value of –20 is used; if it is greater than 19, a value of 19 is used. Only the super-user may use negative nice values.

RETURN VALUES

Since **getpriority( )** can legitimately return the value –1, it is necessary to clear the external variable **errno** prior to the call, then check it afterward to determine if a –1 is an error or a legitimate value.

setpriority( ) returns:

0 on success.

–1 on failure and sets **errno** to indicate the error.

ERRORS

getpriority( ) and **setpriority( )** may set **errno** to:

EINVAL *which* was not one of **PRIO_PROCESS**, **PRIO_PGRP**, or **PRIO_USER**.

ESRCH No process was located using the *which* and *who* values specified.

In addition to the errors indicated above, **setpriority( )** may fail with one of the following errors returned:

EACCES The call to **setpriority( )** would have changed a process' nice value to a value lower than its current value, and the effective user ID of the process executing the call was not that of the super-user.

EPERM A process was located, but neither its effective nor real user ID matched the effective user ID of the caller, and neither the effective nor the real user ID of the process executing **setpriority( )** was super-user.

SEE ALSO

nice(1), **ps**(1), **fork**(2V), **nice**(3v) **renice**(8)

BUGS

It is not possible for the process executing **setpriority( )** to lower any other process down to its current nice value, without requiring super-user privileges.

These system calls are misnamed. They get and set the nice value, not the kernel scheduling priority. **nice**(1) discusses the relationship between nice value and scheduling priority.

NAME

getrlimit, setrlimit – control maximum system resource consumption

SYNOPSIS

```
#include <sys/time.h>
#include <sys/resource.h>

int getrlimit(resource, rlp)
int resource;
struct rlimit *rlp;

int setrlimit(resource, rlp)
int resource;
struct rlimit *rlp;
```

DESCRIPTION

Limits on the consumption of system resources by the current process and each process it creates may be obtained with the **getrlimit( )** call, and set with the **setrlimit( )** call.

The *resource* parameter is one of the following:

RLIMIT_CPU	the maximum amount of cpu time (in seconds) to be used by each process.
RLIMIT_FSIZE	the largest size, in bytes, of any single file that may be created.
RLIMIT_DATA	the maximum size, in bytes, of the data segment for a process; this defines how far a program may extend its break with the **sbrk( )** (see **brk**(2)) system call.
RLIMIT_STACK	the maximum size, in bytes, of the stack segment for a process; this defines how far a program's stack segment may be extended automatically by the system.
RLIMIT_CORE	the largest size, in bytes, of a **core** file that may be created.
RLIMIT_RSS	the maximum size, in bytes, to which a process's resident set size may grow. This imposes a limit on the amount of physical memory to be given to a process; if memory is tight, the system will prefer to take memory from processes that are exceeding their declared resident set size.
RLIMIT_NOFILE	one more than the maximum value that the system may assign to a newly created descriptor. This limit constrains the number of descriptors that a process may create.

A resource limit is specified as a soft limit and a hard limit. When a soft limit is exceeded a process may receive a signal (for example, if the cpu time is exceeded), but it will be allowed to continue execution until it reaches the hard limit (or modifies its resource limit). The **rlimit** structure is used to specify the hard and soft limits on a resource,

```
struct rlimit {
        int     rlim_cur;       /* current (soft) limit */
        int     rlim_max;       /* hard limit */
};
```

Only the super-user may raise the maximum limits. Other users may only alter **rlim_cur** within the range from 0 to **rlim_max** or (irreversibly) lower **rlim_max**.

An "infinite" value for a limit is defined as **RLIM_INFINITY (0x7fffffff).**

Because this information is stored in the per-process information, this system call must be executed directly by the shell if it is to affect all future processes created by the shell; **limit** is thus a built-in command to **csh**(1).

The system refuses to extend the data or stack space when the limits would be exceeded in the normal way: a **brk()** or **sbrk()** call will fail if the data space limit is reached, or the process will be sent a SIGSEGV when the stack limit is reached which will kill the process unless SIGSEGV is handled on a separate signal stack (since the stack cannot be extended, there is no way to send a signal!).

A file I/O operation that would create a file that is too large generates a signal SIGXFSZ; this normally terminates the process, but may be caught. When the soft CPU time limit is exceeded, a signal SIGXCPU is sent to the offending process.

RETURN VALUES

getrlimit() and **setrlimit()** return:

0 on success.

−1 on failure and set **errno** to indicate the error.

ERRORS

EFAULT The address specified by *rlp* was invalid.

EINVAL An invalid *resource* was specified.

In addition to the above, **setrlimit()** may set **errno** to:

EINVAL The new **rlim_cur** exceeds the new **rlim_max**.

EPERM The limit specified would have raised the maximum limit value, and the caller was not the super-user.

SEE ALSO

csh(1), **sh**(1), **brk**(2), **getdtablesize**(2), **quotactl**(2)

BUGS

There should be **limit** and **unlimit** commands in **sh**(1) as well as in **csh**(1).

NAME

getrusage – get information about resource utilization

SYNOPSIS

```
#include <sys/time.h>
#include <sys/resource.h>

int getrusage(who, rusage)
int who;
struct rusage *rusage;
```

DESCRIPTION

getrusage( ) returns information about the resources utilized by the current process, or all its terminated child processes. The interpretation for some values reported, such as **ru_idrss**, are dependent on the clock tick interval. This interval is an implementation dependent value; for example, on Sun-3 sytems the clock tick interval is 1/50 of a second, while on Sun-4 systems the clock tick interval is 1/100 of a second.

The *who* parameter is one of **RUSAGE_SELF** or **RUSAGE_CHILDREN**. The buffer to which *rusage* points will be filled in with the following structure:

```
struct   rusage {
         struct timeval ru_utime;      /* user time used */
         struct timeval ru_stime;      /* system time used */
         int     ru_maxrss;            /* maximum resident set size */
         int     ru_ixrss;             /* currently 0 */
         int     ru_idrss;             /* integral resident set size */
         int     ru_isrss;             /* currently 0 */
         int     ru_minflt;            /* page faults not requiring physical I/O */
         int     ru_majflt;            /* page faults requiring physical I/O */
         int     ru_nswap;             /* swaps */
         int     ru_inblock;           /* block input operations */
         int     ru_oublock;           /* block output operations */
         int     ru_msgsnd;            /* messages sent */
         int     ru_msgrcv;            /* messages received */
         int     ru_nsignals;          /* signals received */
         int     ru_nvcsw;             /* voluntary context switches */
         int     ru_nivcsw;            /* involuntary context switches */
};
```

The fields are interpreted as follows:

ru_utime The total amount of time spent executing in user mode. Time is given in seconds and microseconds.

ru_stime The total amount of time spent executing in system mode. Time is given in seconds and microseconds.

ru_maxrss The maximum resident set size. Size is given in pages (the size of a page, in bytes, is given by the **getpagesize**(2) system call). Also, see WARNINGS.

ru_ixrss Currently returns 0.

ru_idrss An "integral" value indicating the amount of memory in use by a process while the process is running. This value is the sum of the resident set sizes of the process running when a clock tick occurs. The value is given in pages times clock ticks. Note: it does not take sharing into account. Also, see WARNINGS.

ru_isrss Currently returns 0.

ru_minflt The number of page faults serviced which did not require any physical I/O activity. Also, see WARNINGS.

ru_majflt The number of page faults serviced which required physical I/O activity. This could include page ahead operations by the kernel. Also, see WARNINGS.

ru_nswap The number of times a process was swapped out of main memory.

ru_inblock The number of times the file system had to perform input in servicing a **read**(2V) request.

ru_oublock The number of times the file system had to perform output in servicing a **write**(2V) request.

ru_msgsnd The number of messages sent over sockets.

ru_msgrcv The number of messages received from sockets.

ru_nsignals The number of signals delivered.

ru_nvcsw The number of times a context switch resulted due to a process voluntarily giving up the processor before its time slice was completed (usually to await availability of a resource).

ru_nivcsw The number of times a context switch resulted due to a higher priority process becoming runnable or because the current process exceeded its time slice.

RETURN VALUES

getrusage( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

EFAULT The address specified by the *rusage* argument is not in a valid portion of the process's address space.

EINVAL The *who* parameter is not a valid value.

SEE ALSO

gettimeofday(2), **read**(2V), **wait**(2V), **write**(2V)

WARNINGS

The numbers **ru_inblock** and **ru_oublock** account only for real I/O, and are approximate measures at best. Data supplied by the caching mechanism is charged only to the first process to read and the last process to write the data.

The way resident set size is calculated is an approximation, and could misrepresent the true resident set size.

Page faults can be generated from a variety of sources and for a variety of reasons. The customary cause for a page fault is a direct reference by the program to a page which is not in memory. Now, however, the kernel can generate page faults on behalf of the user, for example, servicing **read**(2V) and **write**(2V) system calls. Also, a page fault can be caused by an absent hardware translation to a page, even though the page is in physical memory.

In addition to hardware detected page faults, the kernel may cause pseudo page faults in order to perform some housekeeping. For example, the kernel may generate page faults, even if the pages exist in physical memory, in order to lock down pages involved in a raw I/O request.

By definition, *major* page faults require physical I/O, while *minor* page faults do not require physical I/O. For example, reclaiming the page from the free list would avoid I/O and generate a minor page fault. More commonly, minor page faults occur during process startup as references to pages which are already in

memory. For example, if an address space faults on some "hot" executable or shared library, this results in a minor page fault for the address space. Also, any one doing a **read**(2V) or **write**(2V) to something that is in the page cache will get a minor page fault(s) as well.

BUGS

There is no way to obtain information about a child process which has not yet terminated.

NAME

getsockname – get socket name

SYNOPSIS

```
getsockname(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

DESCRIPTION

getsockname( ) returns the current *name* for the specified socket. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes).

DIAGNOSTICS

A 0 is returned if the call succeeds, –1 if it fails.

ERRORS

The call succeeds unless:

EBADF	*s* is not a valid descriptor.
EFAULT	*name* points to memory not in a valid part of the process address space.
ENOBUFS	Insufficient resources were available in the system to perform the operation.
ENOTSOCK	*s* is a file, not a socket.

SEE ALSO

bind(2), **getpeername**(2), **socket**(2)

BUGS

Names bound to sockets in the UNIX domain are inaccessible; **getsockname( )** returns a zero length name.

NAME

getsockopt, setsockopt – get and set options on sockets

SYNOPSIS

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

int getsockopt(s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int *optlen;

int setsockopt(s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int optlen;
```

DESCRIPTION

getsockopt( ) and **setsockopt( )** manipulate *options* associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the "socket" level, *level* is specified as **SOL_SOCKET**. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* should be set to the protocol number of TCP; see **getprotoent**(3N).

The parameters *optval* and *optlen* are used to access option values for **setsockopt( )**. For **getsockopt( )** they identify a buffer in which the value for the requested option(s) are to be returned. For **getsockopt( )**, *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, *optval* may be supplied as 0.

optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file **<sys/socket.h>** contains definitions for "socket" level options, described below. Options at other protocol levels vary in format and name; consult the appropriate entries in section (4P).

Most socket-level options take an *int* parameter for *optval*. For **setsockopt( )**, the parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled. **SO_LINGER** uses a **struct linger** parameter, defined in **<sys/socket.h>**, which specifies the desired state of the option and the linger interval (see below).

The following options are recognized at the socket level. Except as noted, each may be examined with **getsockopt( )** and set with **setsockopt( )**.

SO_DEBUG	toggle recording of debugging information
SO_REUSEADDR	toggle local address reuse
SO_KEEPALIVE	toggle keep connections alive
SO_DONTROUTE	toggle routing bypass for outgoing messages
SO_LINGER	linger on close if data present
SO_BROADCAST	toggle permission to transmit broadcast messages
SO_OOBINLINE	toggle reception of out-of-band data in band
SO_SNDBUF	set buffer size for output
SO_RCVBUF	set buffer size for input
SO_TYPE	get the type of the socket (get only)
SO_ERROR	get and clear error on the socket (get only)

SO_DEBUG enables debugging in the underlying protocol modules. **SO_REUSEADDR** indicates that the rules used in validating addresses supplied in a **bind**(2) call should allow reuse of local addresses. **SO_KEEPALIVE** enables the periodic transmission of messages on a connected socket. Should the

connected party fail to respond to these messages, the connection is considered broken. A process attempting to write to the socket receives a SIGPIPE signal and the write operation returns an error. By default, a process exits when it receives SIGPIPE. A read operation on the socket returns an error but does not generate SIGPIPE. If the process is waiting in select(2) when the connection is broken, select( ) returns true for any read or write events selected for the socket. SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messags are queued on socket and a close(2V) is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the close( ) attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the setsockopt( ) call when SO_LINGER is requested). If SO_LINGER is disabled and a close( ) is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with recv( ) or read( ) calls without the MSG_OOB flag. SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values. Finally, SO_TYPE and SO_ERROR are options used only with getsockopt( ). SO_TYPE returns the type of the socket, such as SOCK_STREAM; it is useful for servers that inherit sockets on startup. SO_ERROR returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

RETURN VALUES

getsockopt( ) and **setsockopt( )** return:

0 on success.

−1 on failure and set **errno** to indicate the error.

ERRORS

EBADF	*s* is not a valid descriptor.
EFAULT	The address pointed to by *optval* is not in a valid part of the process address space.
ENOPROTOOPT	The option is unknown at the level indicated.
ENOTSOCK	*s* is a file, not a socket.

In addition to the above, **getsockopt( )** may set **errno** to:

EFAULT	*optlen* is not in a valid part of the process address space.

SEE ALSO

ioctl(2), **socket**(2), **getprotoent**(3N)

BUGS

Several of the socket options should be handled at lower levels of the system.

NAME

gettimeofday, settimeofday – get or set the date and time

SYNOPSIS

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

int gettimeofday(tp, tzp)
struct timeval *tp;
struct timezone *tzp;

int settimeofday(tp, tzp)
struct timeval *tp;
struct timezone *tzp;
```

DESCRIPTION

The system's notion of the current Greenwich time and the current time zone is obtained with the **gettimeofday( )** call, and set with the **settimeofday( )** call. The current time is expressed in elapsed seconds and microseconds since 00:00 GMT, January 1, 1970 (zero hour). The resolution of the system clock is hardware dependent; the time may be updated continuously, or in "ticks."

tp points to a **timeval** structure, which includes the following members:

```
long tv_sec;  /* seconds since Jan. 1, 1970 */
long tv_usec; /* and microseconds */
```

If *tp* is a NULL pointer, the current time information is not returned or set.

tzp points to a **timezone( )** structure, which includes the following members:

```
int tz_minuteswest; /* of Greenwich */
int tz_dsttime;     /* type of dst correction to apply */
```

The **timezone( )** structure indicates the local time zone (measured in minutes westward from Greenwich), and a flag that indicates the type of Daylight Saving Time correction to apply. Note: this flag does *not* indicate whether Daylight Saving Time is currently in effect.

Also note that the offset of the local time zone from GMT may change over time, as may the rules for Daylight Saving Time correction. The **localtime( )** routine (see **ctime**(3V)) obtains this information from a file rather than from **gettimeofday( )**. Programs should use **localtime( )** to convert dates and times; the **timezone( )** structure is filled in by **gettimeofday( )** for backward compatibility with existing programs.

The flag indicating the type of Daylight Saving Time correction should have one of the following values (as defined in **<sys/time.h>**):

0	**DST_NONE**: Daylight Savings Time not observed
1	**DST_USA**: United States DST
2	**DST_AUST**: Australian DST
3	**DST_WET**: Western European DST
4	**DST_MET**: Middle European DST
5	**DST_EET**: Eastern European DST
6	**DST_CAN**: Canadian DST
7	**DST_GB**: Great Britain and Eire DST
8	**DST_RUM**: Rumanian DST
9	**DST_TUR**: Turkish DST
10	**DST_AUSTALT**: Australian-style DST with shift in 1986

If *tzp* is a NULL pointer, the time zone information is not returned or set.

Only the super-user may set the time of day or the time zone.

RETURN VALUES

gettimeofday( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

EFAULT	An argument address referenced invalid memory.
EPERM	A user other than the super-user attempted to set the time or time zone.

SEE ALSO

date(1V), **adjtime**(2), **ctime**(3V)

BUGS

Time is never correct enough to believe the microsecond values. There should a mechanism by which, at least, local clusters of systems might synchronize their clocks to millisecond granularity.

NAME

getuid, geteuid – get user identity

SYNOPSIS

int getuid()

int geteuid()

SYSTEM V SYNOPSIS

#include <sys/types.h>

uid_t getuid()

uid_t geteuid()

DESCRIPTION

getuid() returns the real user ID of the current process, **geteuid()** the effective user ID.

The real user ID identifies the person who is logged in. The effective user ID gives the process different permissions during execution of "set-user-ID" mode processes, which use **getuid()** to determine the real-user-id of the process that invoked them.

SEE ALSO

getgid(2V), **setreuid**(2)

NAME

ioctl – control device

SYNOPSIS

int ioctl(fd, request, arg)
int fd, request;
caddr_t arg;

DESCRIPTION

ioctl() performs a special function on the object referred to by the open descriptor *fd*. The set of functions that may be performed depends on the object that *fd* refers to. For example, many operating characteristics of character special files (for instance, terminals) may be controlled with **ioctl()** requests. The writeups in section 4 discuss how **ioctl()** applies to various objects.

The *request* codes for particular functions are specified in include files specific to objects or to families of objects; the writeups in section 4 indicate which include files specify which *request*s.

For most **ioctl()** functions, *arg* is a pointer to data to be used by the function or to be filled in by the function. Other functions may ignore *arg* or may treat it directly as a data item; they may, for example, be passed an **int** value.

RETURN VALUES

ioctl() returns 0 on success for most requests. Some specialized requests may return non-zero values on success; see the description of the request in the man page for the object. On failure, **ioctl()** returns –1 and sets **errno** to indicate the error.

ERRORS

EBADF	*fd* is not a valid descriptor.
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.
ENOTTY	The specified request does not apply to the kind of object to which the descriptor *fd* refers.

ioctl() will also fail if the object on which the function is being performed detects an error. In this case, an error code specific to the object and the function will be returned.

SEE ALSO

execve(2V), **fcntl**(2V), **filio**(4), **mtio**(4), **sockio**(4), **streamio**(4), **termio**(4)

NAME

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

SYNOPSIS

```
#include <signal.h>

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

SYSTEM V SYNOPSIS

```
#include <signal.h>

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

DESCRIPTION

kill() sends the signal *sig* 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*. *sig* may be one of the signals specified in **sigvec**(2), or it may be 0, in which case error checking is performed but no signal is actually sent. This can be used to check the validity of *pid* or the existence of process *pid*.

The real or effective user ID of the sending process must match the real or saved set-user ID of the receiving process, unless the effective user ID of the sending process is super-user. A single exception is the signal **SIGCONT**, which may always be sent to any member of the same session as the current process.

In the following discussion, "system processes" are processes, such as processes 0 and 2, that are not running a regular user program.

If *pid* is greater than zero, the signal is sent to the process whose process ID is equal to *pid*. *pid* may equal 1.

If *pid* is 0, the signal is sent to all processes, except system processes and process 1, whose process group ID is equal to the process group ID of the sender; this is a variant of **killpg**(2).

If *pid* is −1 and the effective user ID of the sender is not super-user, the signal is sent to all processes, except system processes, process 1, and the process sending the signal, whose real or saved set-user ID matches the real or effective ID of the sender.

If *pid* is −1 and the effective user ID of the sender is super-user, the signal is sent to all processes except system processes, process 1, and the process sending the signal.

If *pid* is negative but not −1, the signal is sent to all processes, except system processes, process 1, and the process sending the signal, whose process group ID is equal to the absolute value of *pid*; this is a variant of **killpg**(2).

Processes may send signals to themselves.

SYSTEM V DESCRIPTION

If a signal is sent to a group of processes (as with, if *pid* is 0 or negative), and if the process sending the signal is a member of that group, the signal is sent to that process as well.

The signal **SIGKILL** cannot be sent to process 1.

RETURN VALUES

kill() returns:

0	on success.
−1	on failure and sets **errno** to indicate the error.

ERRORS

kill() will fail and no signal will be sent if any of the following occur:

EINVAL	*sig* was not a valid signal number.
EPERM	The effective user ID of the sending process was not super-user, and neither its real nor effective user ID matched the real or saved set-user ID of the receiving process.
ESRCH	No process could be found corresponding to that specified by *pid*.

SYSTEM V ERRORS

kill() will also fail, and no signal will be sent, if the following occurs:

EINVAL	*sig* is SIGKILL and *pid* is 1.

SEE ALSO

getpid(2V), **killpg**(2), **getpgrp**(2V), **sigvec**(2), **termio**(4)

NAME

killpg – send signal to a process group

SYNOPSIS

int killpg(pgrp, sig)
int pgrp, sig;

DESCRIPTION

killpg() sends the signal *sig* to the process group *pgrp*. See **sigvec**(2) for a list of signals.

The real or effective user ID of the sending process must match the real or saved set-user ID of the receiving process, unless the effective user ID of the sending process is super-user. A single exception is the signal **SIGCONT**, which may always be sent to any descendant of the current process.

RETURN VALUES

killpg() returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

killpg() will fail and no signal will be sent if any of the following occur:

EINVAL *sig* was not a valid signal number.

EPERM The effective user ID of the sending process was not super-user, and neither its real nor effective user ID matched the real or saved set-user ID of one or more of the target processes.

ESRCH No processes were found in the specified process group.

SEE ALSO

kill(2V), **getpgrp**(2V), **sigvec**(2)

NAME

link – make a hard link to a file

SYNOPSIS

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

DESCRIPTION

path1 points to a pathname naming an existing file. *path2* points to a pathname naming a new directory entry to be created. **link()** atomically creates a new link for the existing file and increments the link count of the file by one. {LINK_MAX} (see **pathconf**(2V)) specifies the maximum allowed number of links to the file.

With hard links, both files must be on the same file system. Both the old and the new link share equal access and rights to the underlying object. The super-user may make multiple links to a directory. Unless the caller is the super-user, the file named by *path1* must not be a directory.

Upon successful completion, **link()** marks for update the **st_ctime** field of the file. Also, the **st_ctime** and **st_mtime** fields of the directory that contains the new entry are marked for update.

RETURN VALUES

link() returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

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

EACCES — Search permission is denied for a component of the path prefix pointed to by *path1* or *path2*.

The requested link requires writing in a directory for which write permission is denied.

EDQUOT — The directory in which the entry for the new link is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.

EEXIST — The link referred to by *path2* exists.

EFAULT — One of the path names specified is outside the process's allocated address space.

EIO — An I/O error occurred while reading from or writing to the file system to make the directory entry.

ELOOP — Too many symbolic links were encountered in translating the pathname pointed to by *path1* or *path2*.

EMLINK — The number of links to the file named by *path1* would exceed {LINK_MAX} (see **pathconf**(2V)).

ENAMETOOLONG — The length of the path argument exceeds {PATH_MAX}.

A pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).

ENOENT — A component of the path prefix pointed to by *path1* or *path2* does not exist.

The file referred to by *path1* does not exist.

ENOSPC — The directory in which the entry for the new link is being placed cannot be extended because there is no space left on the file system containing the directory.

ENOTDIR — A component of the path prefix of *path1* or *path2* is not a directory.

EPERM	The file named by *path1* is a directory and the effective user ID is not super-user.
EROFS	The requested link requires writing in a directory on a read-only file system.
EXDEV	The link named by *path2* and the file named by *path1* are on different file systems.

SYSTEM V ERRORS

In addition to the above, the following may also occur:

ENOENT	*path1* or *path2* points to an empty string.

SEE ALSO

symlink(2), **unlink**(2V)

NAME

listen – listen for connections on a socket

SYNOPSIS

int listen(s, backlog)
int s, backlog;

DESCRIPTION

To accept connections, a socket is first created with **socket**(2), a backlog for incoming connections is specified with **listen( )** and then the connections are accepted with **accept**(2). The **listen( )** call applies only to sockets of type **SOCK_STREAM** or **SOCK_SEQPACKET**.

The *backlog* parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client will receive an error with an indication of ECONNREFUSED.

RETURN VALUES

listen( ) returns:

0	on success.
−1	on failure and sets **errno** to indicate the error.

ERRORS

EBADF	*s* is not a valid descriptor.
ENOTSOCK	*s* is not a socket.
EOPNOTSUPP	The socket is not of a type that supports **listen( )**.

SEE ALSO

accept(2), **connect**(2), **socket**(2)

BUGS

The *backlog* is currently limited (silently) to 5.

NAME

lseek, tell – move read/write pointer

SYNOPSIS

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

off_t lseek(fd, offset, whence)
int fd;
off_t offset;
int whence;

long tell(fd)
int fd;
```

DESCRIPTION

lseek() sets the seek pointer associated with the open file or device referred to by the descriptor *fd* according to the value supplied for *whence*. *whence* must be one of the following constants defined in **<unistd.h>**:

SEEK_SET
SEEK_CUR
SEEK_END

If *whence* is **SEEK_SET**, the seek pointer is set to *offset* bytes. If *whence* is **SEEK_CUR**, the seek pointer is set to its current location plus *offset*. If *whence* is **SEEK_END**, the seek pointer is set to the size of the file plus *offset*.

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

The obsolete function **tell(fd)** is equivalent to **lseek(fd, 0L, SEEK_CUR)**.

RETURN VALUES

On success, **lseek()** returns the seek pointer location as measured in bytes from the beginning of the file. On failure, it returns −1 and sets **errno** to indicate the error.

ERRORS

lseek() will fail and the seek pointer will remain unchanged if:

EBADF	*fd* is not an open file descriptor.
EINVAL	*whence* is not a proper value.
	The seek operation would result in an illegal file offset value for the file (for example, a negative file offset for a file other than a character special file).
ESPIPE	*fd* is associated with a pipe or a socket.

SEE ALSO

dup(2V), **open**(2V)

NOTES

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

The constants **L_SET**, **L_INCR**, and **L_XTND** are provided as synonyms for **SEEK_SET**, **SEEK_CUR**, and **SEEK_END**, respectively for backward compatibility but they will disappear in a future release. It is unlikely that the underlying constants 0, 1 and 2 will ever change.

NAME

mctl – memory management control

SYNOPSIS

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

int mctl(addr, len, function, arg)
caddr_t addr;
size_t len;
int function;
void *arg;
```

DESCRIPTION

mctl() applies a variety of control functions over pages identified by the mappings established for the address range [*addr*, *addr* + *len*). The function to be performed is identified by the argument *function*. Legitimate functions are defined in **<sys/mman.h>** as follows.

MC_LOCK — Lock the pages in the range in memory. This function is used to support **mlock**(3). See the **mlock**(3) description for semantics and usage. *arg* is ignored, but must have the value 0.

MC_LOCKAS — Lock the pages in the address space in memory. This function is used to support **mlockall**(3). See the **mlockall**(3) description for semantics and usage. *addr* and *len* are ignored but must be 0. *arg* is an integer built from the flags:

```
#define MCL_CURRENT     0x1     /* lock current mappings */
#define MCL_FUTURE      0x2     /* lock future mappings */
```

MC_SYNC — Synchronize the pages in the range with their backing storage. Optionally invalidate cache copies. This function is used to support **msync**(3). See the **msync**(3) description for semantics and usage. *arg* is used to represent the *flags* argument to **msync**(3).

MC_UNLOCK — Unlock the pages in the range. This function is used to support **mlock**(3). See the **mlock**(3) description for semantics and usage. *arg* is ignored and must have the value 0.

MC_UNLOCKAS — Remove address space memory lock, and locks on all current mappings. This function is used to support **mlockall**(3). *addr* and *len* must have the value 0. *arg* is ignored and must have the value 0.

RETURN VALUES

mctl() returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

EAGAIN — *function* was MC_LOCK or MC_LOCKAS and some or all of the memory identified by the operation could not be locked due to insufficient system resources.

EINVAL — *addr* was not a multiple of the page size as returned by **getpagesize**(2).

addr and/or *len* did not have the value 0 when MC_LOCKAS or MC_UNLOCKAS were specified.

arg was not valid for the function specified.

ENOMEM — Addresses in the range [*addr*, *addr* + *len*) are invalid for the address space of a process, or specify one or more pages which are not mapped.

EPERM — The process's effective user ID was not super-user and one of MC_LOCK, MC_LOCKAS, MC_UNLOCK, or MC_UNLOCKAS was specified.

SEE ALSO

madvise(3), mlock(3), mlockall(3), mmap(2), msync(3)

NAME

mincore – determine residency of memory pages

SYNOPSIS

int mincore(addr, len, vec)
caddr_t addr; int len; result char *vec;

DESCRIPTION

mincore( ) returns the primary memory residency status of pages in the address space covered by mappings in the range [*addr*, *addr* + *len*). The status is returned as a char-per-page in the character array referenced by **vec* (which the system assumes to be large enough to encompass all the pages in the address range). The least significant bit of each character is set to 1 to indicate that the referenced page is in primary memory, 0 if it is not. The settings of other bits in each character is undefined and may contain other information in the future.

RETURN VALUES

mincore( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

mincore( ) will fail if:

EFAULT	A part of the buffer pointer to by *vec* is out-of-range or otherwise inaccessible.
EINVAL	*addr* is not a multiple of the page size as returned by **getpagesize**(2).
ENOMEM	Addresses in the range [*addr*, *addr* + *len*) are invalid for the address space of a process, or specify one or more pages which are not mapped.

SEE ALSO

mmap(2)

NAME

mkdir – make a directory file

SYNOPSIS

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

SYSTEM V SYNOPSIS

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

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

DESCRIPTION

mkdir( ) creates a new directory file with name *path*. The mode mask of the new directory is initialized from *mode*.

The low-order 9 bits of *mode* (the file access permissions) are modified such that all bits set in the process's file mode creation mask are cleared (see **umask**(2V)).

The set-GID bit of *mode* is ignored. The set-GID bit of the new file is inherited from that of the parent directory.

The directory's owner ID is set to the process's effective user ID.

The directory's group ID is set to either:

- the effective group ID of the process, if the filesystem was not mounted with the BSD file-creation semantics flag (see **mount**(2V)) and the set-GID bit of the parent directory is clear, or
- the group ID of the directory in which the file is created.

Upon successful completion, **mkdir( )** marks for update the **st_atime**, **st_ctime**, and **st_mtime** fields of the directory (see **stat**(2V)). The **st_ctime** and **st_mtime** fields of the directory's parent directory are also marked for update.

RETURN VALUES

mkdir( ) returns:

0	on success.
–1	on failure and sets **errno** to indicate the error.

ERRORS

mkdir( ) will fail and no directory will be created if:

EACCES	Search permission is denied for a component of the path prefix of *path*.
	Write permission is denied on the parent directory of the directory to be created.
EDQUOT	The directory in which the entry for the new file is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
	The new directory cannot be created because the user's quota of disk blocks on the file system which will contain the directory has been exhausted.
	The user's quota of inodes on the file system on which the file is being created has been exhausted.
EEXIST	The file referred to by *path* exists.
EFAULT	*path* points outside the process's allocated address space.

EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMLINK	The link count of the parent directory would exceed {LINK_MAX} (see **pathconf**(2V)).
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	A component of the path prefix of *path* does not exist.
ENOSPC	The directory in which the entry for the new file is being placed cannot be extended because there is no space left on the file system containing the directory.
	The new directory cannot be created because there is no space left on the file system which will contain the directory.
	There are no free inodes on the file system on which the file is being created.
ENOTDIR	A component of the path prefix of *path* is not a directory.
EROFS	*path* The parent directory of the directory to be created resides on a read-only file system.

SYSTEM V ERRORS

In addition to the above, the following may also occur:

ENOENT	*path* points to a null pathname.

SEE ALSO

chmod(2V), **mount**(2V), **rmdir**(2V), **stat**(2V), **umask**(2V)

NAME

mknod, mkfifo – make a special file

SYNOPSIS

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

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

int mkfifo(path, mode)
char *path;
mode_t mode;
```

DESCRIPTION

mknod() creates a new file named by the path name pointed to by *path*. The mode of the new file (including file type bits) is initialized from *mode*. The values of the file type bits which are permitted are:

```
#define  S_IFCHR    0020000    /* character special */
#define  S_IFBLK    0060000    /* block special */
#define  S_IFREG    0100000    /* regular */
#define  S_IFIFO    0010000    /* FIFO special */
```

Values of *mode* other than those above are undefined and should not be used.

The access permissions of the mode are modified by the process's mode mask (see **umask**(2V)).

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 either:

- the effective group ID of the process, if the filesystem was not mounted with the BSD file-creation semantics flag (see **mount**(2V)) and the set-gid bit of the parent directory is clear, or
- the group ID of the directory in which the file is created.

If *mode* indicates a block or character special file, *dev* is a configuration dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored.

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

mkfifo() creates a new FIFO special file named by the pathname pointed to by *path*. The access permissions of the new FIFO are initialized from *mode*. The access permissions of *mode* are modified by the process's file creation mask, see **umask**(2V). Bits in *mode* other than the access permissions are ignored.

The FIFO's owner ID is set to the process's effective user ID. The FIFO's group ID is set to the group ID of the directory in which the FIFO is being created or to the process's effective group ID.

Upon successful completion, the **mkfifo()** function marks for update the **st_atime**, **st_ctime**, and **st_mtime** fields of the file. Also, the **st_ctime** and **st_mtime** fields of the directory that contains the new entry are marked for update.

RETURN VALUES

mknod() returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

mkfifo() returns:

0 on success.

−1 on failure and sets **errno** to indicate the error. No FIFO is created.

ERRORS

mknod() fails and the file mode remains unchanged if:

EACCES	Search permission is denied for a component of the path prefix of *path*.
EDQUOT	The directory in which the entry for the new file is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
EDQUOT	The user's quota of inodes on the file system on which the node is being created has been exhausted.
EEXIST	The file referred to by *path* exists.
EFAULT	*path* points outside the process's allocated address space.
EIO	An I/O error occurred while reading from or writing to the file system.
EISDIR	The specified *mode* would have created a directory.
ELOOP	Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} (see **sysconf**(2V)) while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	A component of the path prefix of *path* does not exist.
ENOSPC	The directory in which the entry for the new file is being placed cannot be extended because there is no space left on the file system containing the directory.
ENOSPC	There are no free inodes on the file system on which the file is being created.
ENOTDIR	A component of the path prefix of *path* is not a directory.
EPERM	An attempt was made to create a file of type other than FIFO special and the process's effective user ID is not super-user.
EROFS	The file referred to by *path* resides on a read-only file system.

mkfifo() may set **errno** to:

EACCES	A component of the path prefix denies search permission.
EEXIST	The named file already exists.
ENAMETOOLONG	The length of the path string exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	A component of the path prefix does not exist.
	path points to an empty string.
ENOSPC	The directory that would contain the new file cannot be extended.
	The file system is out of file allocation resources.
ENOTDIR	A component of the path prefix is not a directory.
EROFS	The named file resides on a read-only file system.

SEE ALSO

chmod(2V), **execve**(2V), **pipe**(2V), **stat**(2V), **umask**(2V), **write**(2V)

NAME

mmap – map pages of memory

SYNOPSIS

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

caddr_t mmap(addr, len, prot, flags, fd, off)
caddr_t addr;
size_t len;
int prot, flags, fd;
off_t off;
```

DESCRIPTION

mmap() establishes a mapping between the process's address space at an address *pa* for *len* bytes to the memory object represented by *fd* at *off* for *len* bytes. The value of *pa* is an implementation-dependent function of the parameter *addr* and values of *flags*, further described below. A successful **mmap()** call returns *pa* as its result. The address ranges covered by [*pa*, *pa* + *len*) and [*off*, *off* + *len*) must be legitimate for the *possible* (not necessarily current) address space of a process and the object in question, respectively.

The mapping established by **mmap()** replaces any previous mappings for the process's pages in the range [*pa*, *pa* + *len*).

close(2V) does not unmap pages of the object referred to by a descriptor. Use **munmap**(2) to remove a mapping.

The parameter *prot* determines whether read, write, execute, or some combination of accesses are permitted to the pages being mapped. The protection options are defined in **<sys/mman.h>** as:

```
#define PROT_READ       0x1     /* page can be read */
#define PROT_WRITE      0x2     /* page can be written */
#define PROT_EXEC       0x4     /* page can be executed */
#define PROT_NONE       0x0     /* page can not be accessed */
```

Not all implementations literally provide all possible combinations. **PROT_WRITE** is often implemented as **PROT_READ|PROT_WRITE** and **PROT_EXEC** as **PROT_READ|PROT_EXEC.** However, no implementation will permit a write to succeed where **PROT_WRITE** has not been set. The behavior of **PROT_WRITE** can be influenced by setting **MAP_PRIVATE** in the *flags* parameter, described below.

The parameter *flags* provides other information about the handling of the mapped pages. The options are defined in **<sys/mman.h>** as:

```
#define MAP_SHARED      1       /* Share changes */
#define MAP_PRIVATE     2       /* Changes are private */
#define MAP_TYPE        0xf     /* Mask for type of mapping */
#define MAP_FIXED       0x10    /* Interpret addr exactly */
```

MAP_SHARED and **MAP_PRIVATE** describe the disposition of write references to the memory object. If **MAP_SHARED** is specified, write references will change the memory object. If **MAP_PRIVATE** is specified, the initial write reference will create a private copy of the memory object page and redirect the mapping to the copy. The mapping type is retained across a **fork**(2V).

MAP_FIXED informs the system that the value of *pa* must be *addr*, exactly. The use of **MAP_FIXED** is discouraged, as it may prevent an implementation from making the most effective use of system resources.

When **MAP_FIXED** is not set, the system uses *addr* as a hint in an implementation-defined manner to arrive at *pa*. The *pa* so chosen will be an area of the address space which the system deems suitable for a mapping of *len* bytes to the specified object. All implementations interpret an *addr* value of zero as granting the system complete freedom in selecting *pa*, subject to constraints described below. A non-zero value of *addr* is taken to be a suggestion of a process address near which the mapping should be placed. When the system selects a value for *pa*, it will never place a mapping at address 0, nor will it replace any extant mapping, nor map into areas considered part of the potential data or stack "segments".

The parameter *off* is constrained to be aligned and sized according to the value returned by **getpagesize** (2). When **MAP_FIXED** is specified, the parameter *addr* must also meet these constraints. The system performs mapping operations over whole pages. Thus, while the parameter *len* need not meet a size or alignment constraint, the system will include in any mapping operation any partial page specified by the range [*pa*, *pa* + *len*).

mmap() allows [*pa*, *pa* + *len*) to extend beyond the end of the object, both at the time of the **mmap()** and while the mapping persists, for example if the file was created just prior to the **mmap()** and has no contents, or if the file is truncated. Any reference to addresses beyond the end of the object, however, will result in the delivery of a **SIGBUS** signal.

The system will always zero-fill any partial page at the end of an object. Further, the system will never write out any modified portions of the last page of an object which are beyond its end. References to whole pages following the end of an object will result in a **SIGBUS** signal. **SIGBUS** may also be delivered on various filesystem conditions, including quota exceeded errors.

If the process calls **mlockall**(3) with the **MCL_FUTURE** flag, the pages mapped by all future calls to **mmap()** will be locked in memory. In this case, if not enough memory could be locked, **mmap()** fails and sets **errno** to EAGAIN.

RETURN VALUES

mmap() returns the address at which the mapping was placed (*pa*) on success. On failure, it returns −1 and sets **errno** to indicate the error.

ERRORS

EACCES	*fd* was not open for read and **PROT_READ** or **PROT_EXEC** were specified.
	fd was not open for write and **PROT_WRITE** was specified for a **MAP_SHARED** type mapping.
EAGAIN	Some or all of the mapping could not be locked in memory.
EBADF	*fd* was not open.
EINVAL	The arguments *addr* (if **MAP_FIXED** was specified) and *off* were not multiples of the page size as returned by **getpagesize** (2).
	The **MAP_TYPE** field in *flags* was invalid (neither **MAP_PRIVATE** nor **MAP_SHARED**).
ENODEV	*fd* refered to an object for which **mmap()** is meaningless, such as a terminal.
ENOMEM	**MAP_FIXED** was specified, and the range [*addr*, *addr* + *len*) exceeded that allowed for the address space of a process.
	MAP_FIXED was not specified and there was insufficient room in the address space to effect the mapping.
ENXIO	Addresses in the range [*off*, *off* + *len*) are invalid for *fd*.

SEE ALSO

fork(2V), **getpagesize**(2), **mprotect**(2), **munmap**(2), **mlockall**(3)

NAME

mount – mount file system

SYNOPSIS

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

int mount(type, dir, M_NEWTYPE | flags, data)
char *type;
char *dir;
int flags;
caddr_t data;
```

SYSTEM V SYNOPSIS

```
int mount(spec, dir, rdonly)
char *spec;
char *dir;
int rdonly;
```

DESCRIPTION

mount() attaches a file system to a directory. After a successful return, references to directory *dir* will refer to the root directory on the newly mounted file system. *dir* is a pointer to a null-terminated string containing a path name. *dir* must exist already, and must be a directory. Its old contents are inaccessible while the file system is mounted.

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

The *flags* argument is constructed by the logical OR of the following bits (defined in **<sys/mount.h>**):

M_RDONLY	mount filesystem read-only.
M_NOSUID	ignore set-uid bit on execution.
M_NEWTYPE	this flag must always be set.
M_GRPID	use BSD file-creation semantics (see **open**(2V)).
M_REMOUNT	change options on an existing mount.
M_NOSUB	disallow mounts beneath this filesystem.

Physically write-protected and magnetic tape file systems must be mounted read-only or errors will occur when access times are updated, whether or not any explicit write is attempted.

The *type* string indicates the type of the filesystem. *data* is a pointer to a structure which contains the type specific arguments to mount. Below is a list of the filesystem types supported and the type specific arguments to each:

```
4.2
    struct ufs_args {
            char        *fspec;     /* Block special file to mount */
    };
"lo"
    struct lo_args {
            char        *fsdir;     /* Pathname of directory to mount */
    };
"nfs"
    #include            <nfs/nfs.h>
    #include            <netinet/in.h>
    struct nfs_args {
            struct sockaddr_in  *addr; /* file server address */
            fhandle_t   *fh;        /* File handle to be mounted */
            int         flags;      /* flags */
```

```
                int     wsize;      /* write size in bytes */
                int     rsize;      /* read size in bytes */
                int     timeo;      /* initial timeout in .1 secs */
                int     retrans;    /* times to retry send */
                char    *hostname;  /* server's hostname */
                int     acregmin;   /* attr cache file min secs */
                int     acregmax;   /* attr cache file max secs */
                int     acdirmin;   /* attr cache dir min secs */
                int     acdirmax;   /* attr cache dir max secs */
                char    *netname;   /* server's netname */

        };
    rfs
        struct rfs_args {
                char    *rmtfs      /* name of remote resource */
                struct token {
                        int         t_id;/* token id */
                        char        t_uname[64];/* domain.machine name */
                } *token;           /* Identifier of remote machine */
        };
```

SYSTEM V DESCRIPTION

mount() requests that a file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. *spec* and *dir* point to path names. When **mount()** succeeds, subsequent references to the file named by *dir* refer to the root directory on the mounted file system.

The M_RDONLY bit of *rdonly* is used to control write permission on the mounted file system. If the bit is set, writing is not allowed. Otherwise, writing is permitted according to the access permissions of individual files.

RETURN VALUES

mount() returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

EACCES	Search permission is denied for a component of the path prefix of *dir*.
EBUSY	Another process currently holds a reference to *dir*.
EFAULT	*dir* points outside the process's allocated address space.
ELOOP	Too many symbolic links were encountered in translating the path name of *dir*.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} (see **sysconf**(2V)) while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENODEV	The file system type specified by *type* is not valid or is not configured into the system.
ENOENT	A component of *dir* does not exist.
ENOTDIR	The file named by *dir* is not a directory.
EPERM	The caller is not the super-user.

For a 4.2 file system, **mount()** fails when one of the following occurs:

EACCES	Search permission is denied for a component of the path prefix of *fspec*.

EFAULT	*fspec* points outside the process's allocated address space.
EINVAL	The super block for the file system had a bad magic number or an out of range block size.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating the path name of *fspec*.
EMFILE	No space remains in the mount table.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} (see **sysconf**(2V)) while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	A component of *fspec* does not exist.
ENOMEM	Not enough memory was available to read the cylinder group information for the file system.
ENOTBLK	*fspec* is not a block device.
ENOTDIR	A component of the path prefix of *fspec* is not a directory.
ENXIO	The major device number of *fspec* is out of range (this indicates no device driver exists for the associated hardware).

SYSTEM V ERRORS

EBUSY	The device referred to by *spec* is currently mounted.
	There are no more mount table entries.
ENOENT	The file referred to by *spec* or *dir* does not exist.
ENOTBLK	*spec* is not a block special device.
ENOTDIR	A component of the path prefix of *dir* or *spec* is not a directory.
ENXIO	The device referred to by *spec* does not exist.

SEE ALSO

unmount(2V), **open**(2V), **lofs**(4S), **fstab**(5), **mount**(8)

BUGS

Some of the error codes need translation to more obvious messages.

NAME

mprotect – set protection of memory mapping

SYNOPSIS

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

mprotect(addr, len, prot)
caddr_t addr;
int len, prot;
```

DESCRIPTION

mprotect( ) changes the access protections on the mappings specified by the range [*addr*, *addr* + *len*) to be that specified by *prot*. Legitimate values for *prot* are the same as those permitted for **mmap**(2).

RETURN VALUES

mprotect( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

EACCES *prot* specifies a protection which violates the access permission the process has to the underlying memory object.

EINVAL *addr* is not a multiple of the page size as returned by **getpagesize**(2).

ENOMEM Addresses in the range [*addr*, *addr* + *len*) are invalid for the address space of a process, or specify one or more pages which are not mapped.

When **mprotect( )** fails for reasons other than EINVAL, the protections on some of the pages in the range [*addr*, *addr* + *len*) will have been changed. If the error occurs on some page at address *addr2*, then the protections of all whole pages in the range [*addr*, *addr2*) have been modified.

SEE ALSO

getpagesize(2), **mmap**(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).

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*.

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

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

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

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

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

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

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

Otherwise, the corresponding permissions are denied.

RETURN VALUES

msgctl( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

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

EFAULT *buf* points to an illegal address.

EINVAL *msqid* is not a valid message queue identifier.

cmd is not a valid command.

EPERM *cmd* is equal to **IPC_RMID** or **IPC_SET**. The effective user ID of the calling process is neither super-user, nor the value of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*.

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.

SEE ALSO

intro(2), **msgget**(2), **msgop**(2)

NAME

msgget – get message queue

SYNOPSIS

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

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

DESCRIPTION

msgget( ) returns the message queue identifier associated with **key**.

A message queue identifier and associated message queue and data structure (see **intro**(2)) are created for **key( )** if one of the following is 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-wide standard value of the maximum number of bytes allowed on a message queue.

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

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

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

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

msg_qnum is the number of messages currently on the queue. **msg_qbytes** is the maximum number of bytes allowed on the queue. **msg_lspid** is the process ID of the last process that performed a **msgsnd** operation. **msg_lrpid** is the process ID of the last process that performed a **msgrcv** operation. **msg_stime**

is the time of the last *msgsnd* operation, **msg_rtime** is the time of the last *msgrcv* operation, and **msg_ctime** is the time of the last **msgctl**(2) operation that changed a member of the above structure.

RETURN VALUES

msgget( ) returns A non-negative message queue identifier on success. On failure, it returns −1 and sets **errno** to indicate the error.

ERRORS

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

SEE ALSO

intro(2), **msgctl**(2), **msgop**(2)

NAME

msgop, msgsnd, msgrcv – 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]** (see **msgctl**(2)) *msgp* points to a structure containing the message. This structure is composed of the following members:

```
long    mtype;      /* message type */
char    mtext[1];   /* message text */
```

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

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

- The number of bytes already on the queue is equal to *msg_qbytes* (see **intro**(2)).
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

- If (**msgflg** & IPC_NOWAIT) is "true", the message will not be sent and the calling process will return immediately.
- If (**msgflg** & IPC_NOWAIT) is "false", the calling process will suspend execution until one of the following occurs:
  - The condition responsible for the suspension no longer exists, in which case the message is sent.
  - *msqid* is removed from the system (see **msgctl**(2)). When this occurs, **errno** is set equal to EIDRM, and a value of −1 is returned.
  - The calling process receives a signal that is to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in **signal**(3V).

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[1];   /* 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**(3V).

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

- *msg_qnum* is decremented by 1.
- *msg_lrpid* is set equal to the process ID of the calling process.
- *msg_rtime* is set equal to the current time.

RETURN VALUES

msgsnd( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

msgrcv( ) returns the number of bytes actually placed into *mtext* on success. On failure, it returns −1 and sets **errno** to indicate the error.

ERRORS

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

EACCES	Operation permission is denied to the calling process (see **intro**(2)).
EAGAIN	The message cannot be sent for one of the reasons cited above and (**msgflg** & **IPC_NOWAIT**) is "true".
EFAULT	*msgp* points to an illegal address.
EIDRM	The message queue referred to by *msqid* was removed from the system.
EINTR	The call was interrupted by the delivery of a signal.

EINVAL	*msqid* is not a valid message queue identifier.
	mtype is less than 1.
	msgsz is less than zero or greater than the system-imposed limit.

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

E2BIG	*mtext* is greater than *msgsz* and (**msgflg** & **MSG_NOERROR**) is "false".
EACCES	Operation permission is denied to the calling process.
EFAULT	*msgp* points to an illegal address.
EIDRM	The message queue referred to by *msqid* was removed from the system.
EINTR	The call was interrupted by the delivery of a signal.
EINVAL	*msqid* is not a valid message queue identifier.
	msgsz is less than 0.
ENOMSG	The queue does not contain a message of the desired type and (**msgtyp** & **IPC_NOWAIT**) is "true".

SEE ALSO

intro(2), **msgctl**(2), **msgget**(2), **signal**(3V)

NAME

msync – synchronize memory with physical storage

SYNOPSIS

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

int msync(addr, len, flags)
caddr_t addr;
int len, flags;
```

DESCRIPTION

msync() writes all modified copies of pages over the range [*addr*, *addr* + *len*) to their permanent storage locations. **msync()** optionally invalidates any copies so that further references to the pages will be obtained by the system from their permanent storage locations.

Values for *flags* are defined in **<sys/mman.h>** as:

```
#define MS_ASYNC        0x1     /* Return immediately */
#define MS_INVALIDATE   0x2     /* Invalidate mappings */
```

and are used to control the behavior of **msync()**. One or more flags may be specified in a single call.

MS_ASYNC returns **msync()** immediately once all I/O operations are scheduled; normally, **msync()** will not return until all I/O operations are complete. MS_INVALIDATE invalidates all cached copies of data from memory objects, requiring them to be re-obtained from the object's permanent storage location upon the next reference.

msync() should be used by programs which require a memory object to be in a known state, for example in building transaction facilities.

RETURN VALUES

msync() returns:

0	on success.
−1	on failure and sets **errno** to indicate the error.

ERRORS

EINVAL	*addr* is not a multiple of the current page size.
	len is negative.
	One of the flags MS_ASYNC or MS_INVALID is invalid.
EIO	An I/O error occurred while reading from or writing to the file system.
ENOMEM	Addresses in the range [*addr*, *addr* + *len*) are outside the valid range for the address space of a process.

NAME

munmap – unmap pages of memory.

SYNOPSIS

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

int munmap(addr, len)
caddr_t addr;
int len;
```

DESCRIPTION

munmap() removes the mappings for pages in the range [*addr*, *addr* + *len*). Further references to these pages will result in the delivery of a SIGSEGV signal to the process, unless these pages are considered part of the "data" or "stack" segments.

brk() and **mmap()** often perform implicit **munmap**'s.

RETURN VALUES

munmap() returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

EINVAL *addr* is not a multiple of the page size as returned by **getpagesize**(2).

Addresses in the range [*addr*, *addr* + *len*) are outside the valid range for the address space of a process.

SEE ALSO

brk(2), **getpagesize**(2), **mmap**(2)

NAME

nfssvc, async_daemon – NFS daemons

SYNOPSIS

nfssvc (sock)
int sock;

async_daemon()

DESCRIPTION

nfssvc() starts an NFS daemon listening on socket *sock*. The socket must be **AF_INET**, and **SOCK_DGRAM** (protocol **UDP/IP**). The system call will return only if the socket is invalid.

async_daemon() implements the NFS daemon that handles asynchronous I/O for an NFS client. This system call never returns.

Both system calls result in kernel-only processes with user memory discarded.

SEE ALSO

mountd(8C)

BUGS

There should be a way to dynamically create kernel-only processes instead of having to make system calls from userland to simulate this.

NAME

open – open or create a file for reading or writing

SYNOPSIS

```
#include <fcntl.h>

int open(path, flags[ , mode ] )
char *path;
int flags;
int mode;
```

SYSTEM V SYNOPSIS

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

int open(path, flags[ , mode ] )
char *path;
int flags;
mode_t mode;
```

DESCRIPTION

path points to the pathname of a file. **open()** opens the named file for reading and/or writing, as specified by the *flags* argument, and returns a descriptor for that file. The *flags* argument may indicate the file is to be created if it does not already exist (by specifying the O_CREAT flag), in which case the file is created with mode *mode* as described in **chmod**(2V) and modified by the process' umask value (see **umask**(2V)). If the path is an empty string, the kernel maps this empty pathname to '.', the current directory. *flags* values are constructed by ORing flags from the following list (one and only one of the first three flags below must be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY When opening a FIFO (named pipe – see **mknod**(2V)) with O_RDONLY or O_WRONLY set:

If O_NDELAY is set:

> An **open()** for reading-only returns without delay. An **open()** for writing-only returns an error if no process currently has the file open for reading.

If O_NDELAY is clear:

> A call to **open()** for reading-only blocks until a process opens the file for writing. A call to **open()** for writing-only blocks until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:

> A call to **open()** returns without waiting for carrier.

If O_NDELAY is clear:

> A call to **open()** blocks until carrier is present.

O_NOCTTY When this flag is set, and *path* refers to a terminal device, **open()** prevents the terminal device from becoming the controlling terminal for the process.

O_NONBLOCK Same as O_NDELAY above.

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

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

O_CREAT If the file exists, this flag has no effect. Otherwise, the file is created, and 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 either:

- the effective group ID of the process, if the filesystem was not mounted with the BSD file-creation semantics flag (see **mount**(2V)) and the set-gid bit of the parent directory is clear, or
- the group ID of the directory in which the file is created.

The low-order 12 bits of the file mode are set to the value of *mode*, modified as follows (see **creat**(2V)):

- All bits set in the file mode creation mask of the process are cleared. See **umask**(2V).
- The "save text image after execution" bit of the mode is cleared. See **chmod**(2V).
- The "set group ID on execution" bit of the mode is cleared if the effective user ID of the process is not super-user and the process is not a member of the group of the created file.

O_TRUNC If the file exists and is a regular file, and the file is successfully opened **O_RDWR** or **O_WRONLY**, its length is truncated to zero and the mode and owner are unchanged. **O_TRUNC** has no effect on FIFO special files or directories.

O_EXCL If **O_EXCL** and **O_CREAT** are set, **open()** will fail if the file exists. This can be used to implement a simple exclusive access locking mechanism.

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

The new descriptor is set to remain open across **execve**(2V) system calls; see **close**(2V) and **fcntl**(2V).

There is a system enforced limit on the number of open file descriptors per process, whose value is returned by the **getdtablesize**(2) call.

If **O_CREAT** is set and the file did not previously exist, upon successful completion, **open()** marks for update the **st_atime**, **st_ctime**, and **st_mtime** fields of the file and the **st_ctime** and **st_mtime** fields of the parent directory.

If **O_TRUNC** is set and the file previously existed, upon successful completion, **open()** marks for update the **st_ctime** and **st_mtime** fields of the file.

SYSTEM V DESCRIPTION

If *path* points to an empty string an error results.

The flags above behave as described, with the following exception:

If the **O_NDELAY** or **O_NONBLOCK** flag is set on a call to **open()**, the corresponding flag is set for that file descriptor (see **fcntl**(2V)) and subsequent reads and writes to that descriptor will not block (see **read**(2V) and **write**(2V)).

RETURN VALUES

open() returns a non-negative file descriptor on success. On failure, it returns –1 and sets **errno** to indicate the error.

ERRORS

EACCES	Search permission is denied for a component of the path prefix of *path*.
	The file referred to by *path* does not exist, **O_CREAT** is specified, and the directory in which it is to be created does not permit writing.
	O_TRUNC is specified and write permission is denied for the file named by *path*.
	The required permissions (for reading and/or writing) are denied for the file named by *path*.
EDQUOT	The file does not exist, **O_CREAT** is specified, and the directory in which the entry for the new file is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
	The file does not exist, **O_CREAT** is specified, and the user's quota of inodes on the file system on which the file is being created has been exhausted.
EEXIST	**O_EXCL** and **O_CREAT** were both specified and the file exists.
EFAULT	*path* points outside the process's allocated address space.
EINTR	A signal was caught during the **open()** system call.
EIO	A hangup or error occurred during a STREAMS **open()**.
	An I/O error occurred while reading from or writing to the file system.
EISDIR	The named file is a directory, and the arguments specify it is to be opened for writing.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMFILE	The system limit for open file descriptors per process has already been reached.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENFILE	The system file table is full.
ENOENT	**O_CREAT** is not set and the named file does not exist.
	A component of the path prefix of *path* does not exist.
ENOSPC	The file does not exist, **O_CREAT** is specified, and the directory in which the entry for the new file is being placed cannot be extended because there is no space left on the file system containing the directory.
	The file does not exist, **O_CREAT** is specified, and there are no free inodes on the file system on which the file is being created.
ENOSR	A *stream* could not be allocated.
ENOTDIR	A component of the path prefix of *path* is not a directory.
ENXIO	**O_NDELAY** is set, the named file is a FIFO, **O_WRONLY** is set, and no process has the file open for reading.
	The file is a character special or block special file, and the associated device does not exist.
	O_NONBLOCK is set, the named file is a FIFO, **O_WRONLY** is set, and no process has the file open for reading.
	A STREAMS module or driver open routine failed.
EOPNOTSUPP	An attempt was made to open a socket (not currently implemented).

EROFS — The named file does not exist, **O_CREAT** is specified, and the file system on which it is to be created is a read-only file system.

The named file resides on a read-only file system, and the file is to be opened for writing.

SYSTEM V ERRORS

In addition to the above, the following may also occur:

ENOENT — *path* points to an empty string.

SEE ALSO

chmod(2V), **close**(2V), **creat**(2V), **dup**(2V), **fcntl**(2V), **getdtablesize**(2), **getmsg**(2), **lseek**(2V), **mknod**(2V), **mount**(2V), **putmsg**(2), **read**(2V), **umask**(2V) **write**(2V)

NAME

pathconf, fpathconf – query file system related limits and options

SYNOPSIS

```
#include <unistd.h>

long pathconf(path, name)
char *path;
int name;

long fpathconf(fd, name)
int fd, name;
```

DESCRIPTION

pathconf() and **fpathconf()** provide a method for the application to determine the current value of a configurable limit or option that is associated with a file or directory,

For **pathconf()**, *path* points to the pathname of a file or directory. For **fpathconf()**, *fd* is an open file descriptor.

The convention used throughout sections 2 and 3 is that {LIMIT} means that LIMIT is something that can change from file to file (due to multiple file systems on the same machine). The actual value for LIMIT is typically not defined in any header file since it is not invariant. Instead, pathconf must be called to retrieve the value. **pathconf()** understands a list of flags that are named similarly to the value being queried.

The following table lists the name and meaning of each conceptual limit.

Limit	*Meaning*
{LINK_MAX}	Max links to an object.
{MAX_CANON}	Max tty input line size.
{MAX_INPUT}	Max packet a tty can accept at once.
{NAME_MAX}	Max filename length.
{PATH_MAX}	Max pathname length.
{PIPE_BUF}	Pipe buffer size.
{_POSIX_CHOWN_RESTRICTED}	If true only root can chown() files, otherwise anyone may give away files.
{_POSIX_NO_TRUNC}	If false filenames > {NAME_MAX} are truncated, otherwise an error.
{_POSIX_VDISABLE}	A char to use to disable tty special chars.

The following table lists the name of each limit, the flag passed to **pathconf()** to retrieve the value of each variable, and some notes about usage.

Limit	*Pathconf Flag*	*Notes*
{LINK_MAX}	_PC_LINK_MAX	1
{MAX_CANON}	_PC_MAX_CANON	2
{MAX_INPUT}	_PC_MAX_INPUT	2
{NAME_MAX}	_PC_NAME_MAX	3,4
{PATH_MAX}	_PC_PATH_MAX	4,5
{PIPE_BUF}	_PC_PIPE_BUF	6
{_POSIX_CHOWN_RESTRICTED}	_PC_CHOWN_RESTRICTED	7,8
{_POSIX_NO_TRUNC}	_PC_NO_TRUNC	3,4,8
{_POSIX_VDISABLE}	_PC_VDISABLE	2,8

The following notes apply to the entries in the preceding table.

1 If *path* or *fd* refers to a directory, the value returned applies to the directory itself.

2 The behavior is undefined if *path* or *fd* does not refer to a terminal file.

3 If *path* or *fd* refers to a directory, the value returned applies to the file names within the directory.

4 The behavior is undefined if *path* or *fd* does not refer to a directory.

5 If *path* or *fd* refers to a directory, the value returned is the maximum length of a relative pathname when the specified directory is the working directory.

6 If *path* refers to a FIFO, or *fd* refers to a pipe of FIFO, the value returned applies to the referenced object itself. If *path* or *fd* refers to a directory, the value returned applies to any FIFOs that exist or can be created within the directory. If *path* or *fd* refer to any other type of file, the behavior is undefined.

7 If *path* or *fd* refer to a directory, the value returned applies to any files, other than directories, that exist or can be created within the directory.

8 The option in question is a boolean; the return value is 0 or 1.

RETURN VALUES

On success, **pathconf()** and **fpathconf()** return the current variable value for the file or directory. On failure, they return −1 and set **errno** to indicate the error.

If the variable corresponding to *name* has no limit for the path or file descriptor, **pathconf()** and **fpathconf()** return −1 without changing **errno**.

ERRORS

pathconf() and **fpathconf()** may set **errno** to:

EINVAL The value of name is invalid.

For each of the following conditions, if the condition is detected, **pathconf()** fails and sets **errno** to:

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

EINVAL The implementation does not support an association of the variable name with the specified file.

ENAMETOOLONG The length of the path argument exceeds {PATH_MAX}.

A pathname component is longer than {NAME_MAX} while {POSIX_NO_TRUNC} is in effect.

ENOENT The named file does not exist.

path points to an empty string.

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

For each of the following conditions, if the condition is detected, **fpathconf()** fails and sets **errno** to:

EBADF

The *fd* argument is not a valid file descriptor.

EINVAL The implementation does not support an association of the variable name with the specified file.

NAME

pipe – create an interprocess communication channel

SYNOPSIS

int pipe(fd)
int fd[2];

DESCRIPTION

The **pipe()** system call creates an I/O mechanism called a pipe and returns two file descriptors, *fd*[0] and *fd*[1]. *fd*[0] is opened for reading and *fd*[1] is opened for writing. The **O_NONBLOCK** flag is clear on both file descriptors (see **open**(2V)). When the pipe is written using the descriptor *fd*[1] up to {PIPE_BUF} (see **sysconf**(2V)) bytes of data are buffered before the writing process is blocked. A read only file descriptor *fd*[0] accesses the data written to *fd*[1] on a FIFO (first-in-first-out) basis.

The standard programming model is that after the pipe has been set up, two (or more) cooperating processes (created by subsequent **fork**(2V) calls) will pass data through the pipe using **read**(2V) and **write**(2V).

Read calls on an empty pipe (no buffered data) with only one end (all write file descriptors closed) returns an EOF (end of file).

Pipes are really a special case of the **socketpair**(2) call and, in fact, are implemented as such in the system.

A SIGPIPE signal is generated if a write on a pipe with only one end is attempted.

Upon successful completion, **pipe()** marks for update the **st_atime**, **st_ctime**, and **st_mtime** fields of the pipe.

RETURN VALUES

pipe() returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

EFAULT The array *fd* is in an invalid area of the process's address space.

EMFILE Too many descriptors are active.

ENFILE The system file table is full.

SEE ALSO

sh(1), **fork**(2V), **read**(2V), **socketpair**(2), **write**(2V)

BUGS

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

NAME

poll – I/O multiplexing

SYNOPSIS

```
#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 (see **intro**(2)). **poll()** identifies those file descriptors on which a user can send or receive messages, or on which certain events have occurred. A user can receive messages using **read**(2V) or **getmsg**(2) and can send messages using **write**(2V) and **putmsg**(2). Certain **ioctl**(2) calls, such as **I_RECVFD** and **I_SENDFD** (see **streamio**(4)), can also be used to receive and send messages on streams.

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 ORing any combination of the following event flags:

POLLIN If the file descriptor refers to a stream, 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. If the file descriptor is not a stream, the file descriptor is readable. In **revents**, this flag is mutually exclusive with **POLLPRI**.

POLLPRI If the file descriptor is a stream, a priority message is present on the **stream head** read queue. This flag is set even if the message is of zero length. If the file descriptor is not a stream, some exceptional condition has occurred. In **revents**, this flag is mutually exclusive with **POLLIN**.

POLLOUT If the file descriptor is a stream, the first downstream write queue in the *stream* is not full. Priority control messages can be sent (see **putmsg**(2)) at any time. If the file descriptor is not a stream, it is writable.

POLLERR If the file descriptor is a stream, 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 If the file descriptor is a stream, 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 specify an open file descriptor. 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 the system limit of open files (see **getdtablesize**(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* milliseconds 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.

RETURN VALUES

poll() returns a non-negative value on success. A positive value indicates the total number of file descriptors that has been selected (for instance, file descriptors for which the **revents** field is non-zero). 0 indicates the call timed out and no file descriptors have been selected. On failure, **poll()** returns −1 and sets **errno** to indicate the error.

ERRORS

EAGAIN	Allocation of internal data structures failed, but the 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.
	nfds is greater than the system limit of open files.

SEE ALSO

getdtablesize(2), **getmsg**(2), **intro**(2), **ioctl**(2), **putmsg**(2), **read**(2V), **select**(2), **write**(2V), **streamio**(4)

NAME

profil – execution time profile

SYNOPSIS

```
int profil(buf, bufsiz, offset, scale)
short *buf;
int bufsiz;
void (*offset)();
int scale;
```

DESCRIPTION

profil( ) enables run-time execution profiling, and reserves a buffer for maintaining raw profiling statistics. *buf* points to an area of core of length *bufsiz* (in bytes). After the call to **profil( )**, the user's program counter (pc) is examined at each clock tick (10 milliseconds on Sun-4 systems, 20 milliseconds on Sun-3 systems); *offset* is subtracted from its value, and the result multiplied by *scale*. If the resulting number corresponds to a word within the buffer, that word is incremented.

scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0xffff gives a 1-to-1 mapping of pc values to words in *buf*; 0x7fff maps each pair of instruction words together. 0x2 maps all instructions onto the beginning of *buf* (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 **execve( )** is executed, but remains on in child and parent both after a **fork( )**. Profiling is turned off if an update in *buf* would cause a memory fault.

RETURN VALUES

profil( ) always succeeds and returns 0.

SEE ALSO

gprof(1), **getitimer**(2), **monitor**(3)

NAME

ptrace – process trace

SYNOPSIS

```
#include <signal.h>
#include <sys/ptrace.h>
#include <sys/wait.h>

ptrace(request, pid, addr, data [ , addr2 ] )
enum ptracereq request;
int pid;
char *addr;
int data;
char *addr2;
```

DESCRIPTION

ptrace( ) provides a means by which a process may control the execution of another process, and examine and change its core image. Its primary use is for the implementation of breakpoint debugging. There are five arguments whose interpretation depends on the *request* argument. Generally, *pid* is the process ID of the traced process. A process being traced behaves normally until it encounters some signal whether internally generated like "illegal instruction" or externally generated like "interrupt". See **sigvec**(2) for the list. Then the traced process enters a stopped state and the tracing process is notified using **wait**(2V). When the traced process is in the stopped state, its core image can be examined and modified using **ptrace( )**. If desired, another **ptrace( )** request can then cause the traced process either to terminate or to continue, possibly ignoring the signal.

Note: several different values of the *request* argument can make **ptrace( )** return data values — since −1 is a possibly legitimate value, to differentiate between −1 as a legitimate value and −1 as an error code, you should clear the **errno** global error code before doing a **ptrace( )** call, and then check the value of **errno** afterwards.

The value of the *request* argument determines the precise action of the call:

PTRACE_TRACEME
: This request is the only one used by the traced process; it declares that the process is to be traced by its parent. All the other arguments are ignored. Peculiar results will ensue if the parent does not expect to trace the child.

PTRACE_PEEKTEXT
PTRACE_PEEKDATA
: The word in the traced process's address space at *addr* is returned. If the instruction and data spaces are separate (for example, historically on a PDP-11), request **PTRACE_PEEKTEXT** indicates instruction space while **PTRACE_PEEKDATA** indicates data space. Otherwise, either request may be used, with equal results; *addr* must be a multiple of 4 on a Sun-4 system. The child must be stopped. The input *data* and *addr2* are ignored.

PTRACE_PEEKUSER
: The word of the system's per-process data area corresponding to *addr* is returned. *addr* must be a valid offset within the kernel's per-process data pages. This space contains the registers and other information about the process; its layout corresponds to the *user* structure in the system (see **<sys/user.h>**).

PTRACE_POKETEXT
PTRACE_POKEDATA
: The given *data* are written at the word in the process's address space corresponding to *addr*. *addr* must be a multiple of 4 on a Sun-4 system. No useful value is returned. If the instruction and data spaces are separate, request **PTRACE_PEEKTEXT** indicates instruction space while **PTRACE_PEEKDATA** indicates data space. The **PTRACE_POKETEXT** request must be used to write into a process's text space even if the instruction and data spaces are not separate.

PTRACE_POKEUSER
The process's system data are written, as it is read with request **PTRACE_PEEKUSER**. Only a few locations can be written in this way: the general registers, the floating point and status registers, and certain bits of the processor status word.

PTRACE_CONT
The *data* argument is taken as a signal number and the child's execution continues at location *addr* as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal that caused the stop should be ignored, or that value fetched out of the process's image indicating which signal caused the stop. If *addr* is (int *)1 then execution continues from where it stopped. *addr* must be a multiple of 4 on a Sun-4 system.

PTRACE_KILL
The traced process terminates, with the same consequences as **exit**(2V).

PTRACE_SINGLESTEP
Execution continues as in request **PTRACE_CONT**; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is **SIGTRAP**. On Sun-3 and Sun386i systems, the status register T-bit is used and just one instruction is executed. This is part of the mechanism for implementing breakpoints. On a Sun-4 system this will return an error since there is no hardware assist for this feature. Instead, the user should insert breakpoint traps in the debugged program with **PTRACE_POKETEXT**.

PTRACE_ATTACH
Attach to the process identified by the *pid* argument and begin tracing it. **PTRACE_ATTACH** causes a **SIGSTOP** to be sent to process *pid*. Process *pid* does not have to be a child of the requestor, but the requestor must have permission to send process *pid* a signal and the effective user IDs of the requesting process and process *pid* must match.

PTRACE_DETACH
Detach the process being traced. Process *pid* is no longer being traced and continues its execution. The *data* argument is taken as a signal number and the process continues at location *addr* as if it had incurred that signal.

PTRACE_GETREGS
The traced process's registers are returned in a structure pointed to by the *addr* argument. The registers include the general purpose registers, the program counter and the program status word. The "regs" structure defined in **<machine/reg.h>** describes the data that are returned.

PTRACE_SETREGS
The traced process's registers are written from a structure pointed to by the *addr* argument. The registers include the general purpose registers, the program counter and the program status word. The "regs" structure defined in **reg.h** describes the data that are set.

PTRACE_GETFPREGS
(Sun-3, Sun-4 and Sun386i systems only) The traced process's FPP status is returned in a structure pointed to by the *addr* argument. The status includes the 68881 (80387 on Sun386i systems) floating point registers and the control, status, and instruction address registers. The "fp_status" structure defined in **reg.** describes the data that are returned. The **fp_state** structure defined in **<machine/fp.h>** describes the data that are returned on a Sun386i system.

PTRACE_SETFPREGS
(Sun-3, Sun-4 and Sun386i systems only) The traced process's FPP status is written from a structure pointed to by the *addr* argument. The status includes the FPP floating point registers and the control, status, and instruction address registers. The "fp_status" structure defined in **reg.h** describes the data that are set. The "fp_state" structure defined in **fp.h** describes the data that are returned on a Sun386i system.

PTRACE_GETFPAREGS
(a Sun-3 system with FPA only) The traced process's FPA registers are returned in a structure pointed to by the *addr* argument. The "fpa_regs" structure defined in **reg.h** describes the data that are returned.

PTRACE_SETFPAREGS
(a Sun-3 system with FPA only) The traced process's FPA registers are written from a structure pointed to by the *addr* argument. The "fpa_regs" structure defined in **reg.h** describes the data that are set.

PTRACE_READTEXT
PTRACE_READDATA
Read data from the address space of the traced process. If the instruction and data spaces are separate, request **PTRACE_READTEXT** indicates instruction space while **PTRACE_READDATA** indicates data space. The *addr* argument is the address within the traced process from where the data are read, the *data* argument is the number of bytes to read, and the *addr2* argument is the address within the requesting process where the data are written.

PTRACE_WRITETEXT
PTRACE_WRITEDATA
Write data into the address space of the traced process. If the instruction and data spaces are separate, request **PTRACE_READTEXT** indicates instruction space while **PTRACE_READDATA** indicates data space. The *addr* argument is the address within the traced process where the data are written, the *data* argument is the number of bytes to write, and the *addr2* argument is the address within the requesting process from where the data are read.

PTRACE_SETWRBKPT
(Sun386i systems only) Set a write breakpoint at location *addr* in the process being traced. Whenever a write is directed to this location a breakpoint will occur and a SIGTRAP signal will be sent to the process. The *data* argument specifies which debug register should be used for the address of the breakpoint and must be in the range 0 through 3, inclusive. The *addr2* argument specifies the length of the operand in bytes, and must be one of 1, 2, or 4.

PTRACE_SETACBKPT
(Sun386i systems only) Set an access breakpoint at location *addr* in the process being traced. When location *addr* is read or written a breakpoint will occur and the process will be sent a SIGTRAP signal. The *data* argument specifies which debug register should be used for the address of the breakpoint and must be in the range 0 through 3, inclusive. The *addr2* argument specifies the length of the operand in bytes, and must be one of 1, 2, or 4.

PTRACE_CLRBKPT
(Sun386i systems only) Clears all break points set with **PTRACE_SETACBKPT** or **PTRACE_SETWRBKPT**.

PTRACE_SYSCALL
Execution continues as in request **PTRACE_CONT**; until the process makes a system call. The process receives a **SIGTRAP** signal and stops. At this point the arguments to the system call may be inspected in the process *user* structure using the **PTRACE_PEEKUSER** request. The system call number is available in place of the 8th argument. Continuing with another **PTRACE_SYSCALL** will stop the process again at the completion of the system call. At this point the result of the system call and error value may be inspected in the process *user* structure.

PTRACE_DUMPCORE
Dumps a core image of the traced process to a file. The name of the file is obtained from the *addr* argument.

As indicated, these calls (except for requests **PTRACE_TRACEME**, **PTRACE_ATTACH** and **PTRACE_DETACH**) can be used only when the subject process has stopped. The **wait()** call is used to determine when a process stops; in such a case the "termination" status returned by **wait()** has the value **WSTOPPED** to indicate a stop rather than genuine termination.

To forestall possible fraud, **ptrace()** inhibits the setUID and setGID facilities on subsequent **execve**(2V) calls. If a traced process calls **execve()**, it will stop before executing the first instruction of the new image, showing signal **SIGTRAP**.

On the Sun, "word" also means a 32-bit integer.

RETURN VALUES

On success, the value returned by **ptrace()** depends on *request* as follows:

PTRACE_PEEKTEXT **PTRACE_PEEKDATA**	The word in the traced process's address space at *addr*.
PTRACE_PEEKUSER	The word of the system's per-process data area corresponding to *addr*.

On failure, these requests return −1 and set **errno** to indicate the error.

For all other values of *request*, **ptrace()** returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

EIO	The request code is invalid.
	The given signal number is invalid.
	The specified address is out of bounds.
EPERM	The specified process cannot be traced.
ESRCH	The specified process does not exist.
	request requires process *pid to be* traced by the current process and stopped, and process *pid* is not being traced by the current process.
	request requires process *pid to be* traced by the current process and stopped, and process *pid* is not stopped.

SEE ALSO

adb(1), **intro**(2), **ioctl**(2), **sigvec**(2), **wait**(2V)

BUGS

ptrace() is unique and arcane; it should be replaced with a special file which can be opened and read and written. The control functions could then be implemented with **ioctl**(2) calls on this file. This would be simpler to understand and have much higher performance.

The requests **PTRACE_TRACEME** through **PTRACE_SINGLESTEP** are standard UNIX system **ptrace()** requests. The requests **PTRACE_ATTACH** through **PTRACE_DUMPCORE** and the fifth argument, *addr2*, are unique to SunOS.

The request **PTRACE_TRACEME** should be able to specify signals which are to be treated normally and not cause a stop. In this way, for example, programs with simulated floating point (which use "illegal instruction" signals at a very high rate) could be efficiently debugged.

The error indication, −1, is a legitimate function value; **errno**, (see **intro**(2)), can be used to clarify what it means.

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 that 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 not be a NULL pointer 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 a NULL pointer 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.

RETURN VALUES

putmsg() returns:

0	on success.
–1	on failure and sets **errno** to indicate the error.

ERRORS

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. 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*.
	flags is set to **RS_HIPRI** and no control part was supplied.
	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.
	The control part of the message is larger than the maximum configured size of the control part of a message.
	The data part of the 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.

SEE ALSO

getmsg(2), **intro**(2), **poll**(2), **read**(2V), **write**(2V)

NAME

quotactl – manipulate disk quotas

SYNOPSIS

```
#include <ufs/quota.h>

int quotactl(cmd, special, uid, addr)
int cmd;
char *special;
int uid;
caddr_t addr;
```

DESCRIPTION

The **quotactl()** call manipulates disk quotas. *cmd* indicates a command to be applied to the user ID *uid*. *special* is a pointer to a null-terminated string containing the path name of the block special device for the file system being manipulated. The block special device must be mounted as a UFS file system (see **mount**(2V)). *addr* is the address of an optional, command specific, data structure which is copied in or out of the system. The interpretation of *addr* is given with each command below.

Q_QUOTAON — Turn on quotas for a file system. *addr* points to the path name of file containing the quotas for the file system. The quota file must exist; it is normally created with the **quotacheck**(8) program. This call is restricted to the super-user.

Q_QUOTAOFF — Turn off quotas for a file system. *addr* and *uid* are ignored. This call is restricted to the super-user.

Q_GETQUOTA — Get disk quota limits and current usage for user *uid*. *addr* is a pointer to a **dqblk** structure (defined in **<ufs/quota.h>**). Only the super-user may get the quotas of a user other than himself.

Q_SETQUOTA — Set disk quota limits and current usage for user *uid*. *addr* is a pointer to a **dqblk** structure (defined in **quota.h**). This call is restricted to the super-user.

Q_SETQLIM — Set disk quota limits for user *uid*. *addr* is a pointer to a **dqblk** structure (defined in **quota.h**). This call is restricted to the super-user.

Q_SYNC — Update the on-disk copy of quota usages for a file system. If *special* is null then all file systems with active quotas are sync'ed. *addr* and *uid* are ignored.

RETURN VALUES

quotactl() returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

EFAULT — *addr* or *special* are invalid.

EINVAL — The kernel has not been compiled with the QUOTA option.

cmd is invalid.

ENODEV — *special* is not a mounted UFS file system.

ENOENT — The file specified by *special* or *addr* does not exist.

ENOTBLK — *special* is not a block device.

EPERM — The call is privileged and the caller was not the super-user.

ESRCH — No disc quota is found for the indicated user.

Quotas have not been turned on for this file system.

EUSERS — The quota table is full.

If *cmd* is **Q_QUOTAON** **quotactl()** may set errno to:

EACCES The quota file pointed to by *addr* exists but is not a regular file.

The quota file pointed to by *addr* exists but is not on the file system pointed to by *special*.

EBUSY **Q_QUOTAON** attempted while another **Q_QUOTAON** or **Q_QUOTAOFF** is in progress.

SEE ALSO

quota(1), **getrlimit**(2), **mount**(2V), **quotacheck**(8), **quotaon**(8)

BUGS

There should be some way to integrate this call with the resource limit interface provided by **setrlimit()** and **getrlimit**(2).

Incompatible with Melbourne quotas.

NAME

read, readv – read input

SYNOPSIS

```
int read(fd, buf, nbyte)
int fd;
char *buf;
int nbyte;

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

int readv(fd, iov, iovcnt)
int fd;
struct iovec *iov;
int iovcnt;
```

DESCRIPTION

read() attempts to read *nbyte* bytes of data from the object referenced by the descriptor *fd* into the buffer pointed to by *buf*. **readv()** performs the same action as **read()**, but scatters the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0],*iov*[1], ..., *iov*[*iovcnt* – 1].

If *nbyte* is zero, **read()** takes no action and returns 0. **readv()**, however, returns –1 and sets the global variable **errno** (see ERRORS below).

For **readv()**, the **iovec** structure is defined as

```
struct iovec {
        caddr_t iov_base;
        int     iov_len;
};
```

Each **iovec** entry specifies the base address and length of an area in memory where data should be placed. **readv()** will always fill an area completely before proceeding to the next.

On objects capable of seeking, the **read()** starts at a position given by the pointer associated with *fd* (see **lseek**(2V)). Upon return from **read()**, the pointer is incremented by the number of bytes actually read.

Objects that are not capable of seeking always read from the current position. The value of the pointer associated with such an object is undefined.

Upon successful completion, **read()** and **readv()** return the number of bytes actually read and placed in the buffer. The system guarantees to read the number of bytes requested if the descriptor references a normal file which has that many bytes left before the EOF (end of file), but in no other case.

If the process calling **read()** or **readv()** receives a signal before any data are read, the system call is restarted unless the process explicitly set the signal to interrupt the call using **sigvec()** or **sigaction()** (see the discussions of **SV_INTERRUPT** on **sigvec**(2) and **SA_INTERRUPT** on **sigaction**(3V)). If **read()** or **readv()** is interrupted by a signal after successfully reading some data, it returns the number of bytes read.

If *nbyte* is not zero and **read()** returns 0, then EOF has been reached. If **readv()** returns 0, then EOF has been reached.

A **read()** or **readv()** from a STREAMS file (see **intro**(2)) 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**(2) request (see **streamio**(4)), and can be tested with the **I_GRDOPT ioctl()** request. In byte-stream mode, **read()** and **readv()** will retrieve data from the *stream* until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, **read()** and **readv()** will retrieve data until as many bytes as were requested are transferred, or until a message boundary is reached. If the **read()** or **readv()** does not retrieve all the data in a message, the remaining data are left on the *stream*, and can be retrieved by the

next **read()**, **readv()**, or **getmsg**(2) call. Message-discard mode also retrieves data until as many bytes as were requested are transferred, or a message boundary is reached. However, unread data remaining in a message after the **read()** or **readv()** returns are discarded, and are not available for a subsequent **read()**, **readv()**, or **getmsg()**.

When attempting to read from a descriptor associated with an empty pipe, socket, FIFO, or *stream*:

- If the object the descriptor is associated with is marked for 4.2BSD-style non-blocking I/O (with the **FIONBIO ioctl()** request or a call to **fcntl**(2V) using the **FNDELAY** flag from **<sys/file.h>** or the **O_NDELAY** flag from **<fcntl.h>** in the 4.2BSD environment), the read will return −1 and **errno** will be set to EWOULDBLOCK.
- If the descriptor is marked for System V-style non-blocking I/O (using **fcntl()** with the **FNBIO** flag from **<sys/file.h>** or the **O_NDELAY** flag from **<fcntl.h>** in the System V environment), and does not refer to a *stream*, the read will return 0. Note: this is indistinguishable from EOF.
- If the descriptor is marked for POSIX-style non-blocking I/O (using **fcntl()** with the **O_NONBLOCK** flag from **<fcntl.h>**) and refers to a *stream*, the read will return −1 and **errno** will be set to EAGAIN.
- If neither the descriptor nor the object it refers to are marked for non-blocking I/O, the read will block until data is available to be read or the object has been "disconnected". A pipe or FIFO is "disconnected" when no process has the object open for writing; a socket that was connected is "disconnected" when the connection is broken; a stream is "disconnected" when a hangup condition occurs (for instance, when carrier drops on a terminal).

If the descriptor or the object is marked for non-blocking I/O, and less data are available than are requested by the **read()** or **readv()**, only the data that are available are returned, and the count indicates how many bytes of data were actually read.

When reading from a STREAMS file, handling of zero-byte messages is determined by the current read mode setting. In byte-stream mode, **read()** and **readv()** accept data until as many bytes as were requested are transferred, or until there is no more data to read, or until a zero-byte message block is encountered. **read()** and **readv()** then return the number of bytes read, and places the zero-byte message back on the *stream* to be retrieved by the next **read()**, **readv()**, 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()** or **readv()** 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 **streamhead.**

Upon successful completion, **read()** and **readv()** mark for update the **st_atime** field of the file.

RETURN VALUES

read() and **readv()** return the number of bytes actually read on success. On failure, they return −1 and set **errno** to indicate the error.

ERRORS

EAGAIN	The descriptor referred to a *stream*, was marked for System V-style non-blocking I/O, and no data were ready to be read.
EBADF	*d* is not a valid file descriptor open for reading.
EBADMSG	The message waiting to be read on a *stream* is not a data message.
EFAULT	*buf* points outside the allocated address space.
EINTR	The process performing a read from a slow device received a signal before any data arrived, and the signal was set to interrupt the system call.
EINVAL	The *stream* is linked below a multiplexor.
	The pointer associated with *fd* was negative.

EIO	An I/O error occurred while reading from or writing to the file system.
	The calling process is in a background process group and is attempting to read from its controlling terminal and the process is ignoring or blocking SIGTTIN.
	The calling process is in a background process group and is attempting to read from its controlling terminal and the process is orphaned.
EISDIR	*fd* refers to a directory which is on a file system mounted using the NFS.
EWOULDBLOCK	The file was marked for 4.2BSD-style non-blocking I/O, and no data were ready to be read.

In addition to the above, **readv( )** may set **errno** to:

EFAULT	Part of *iov* points outside the process's allocated address space.
EINVAL	*iovcnt* was less than or equal to 0, or greater than 16.
	One of the **iov_len** values in the *iov* array was negative.
	The sum of the **iov_len** values in the *iov* array overflowed a 32-bit integer.

A **read( )** or **readv( )** 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

dup(2V), **fcntl**(2V), **getmsg**(2), **intro**(2), **ioctl**(2), **lseek**(2V), **open**(2V), **pipe**(2V), **select**(2), **socket**(2), **socketpair**(2), **streamio**(4), **termio**(4)

NAME

readlink – read value of a symbolic link

SYNOPSIS

int readlink(path, buf, bufsiz)
char *path, *buf;
int bufsiz;

DESCRIPTION

readlink() places the contents of the symbolic link referred to by *path* in the buffer *buf* which has size *bufsiz*. The contents of the link are not null terminated when returned.

RETURN VALUES

readlink() returns the number of characters placed in the buffer on success. On failure, it returns −1 and sets **errno** to indicate the error.

ERRORS

readlink() will fail and the buffer will be unchanged if:

EACCES	Search permission is denied for a component of the path prefix of *path*.
EFAULT	*path* or *buf* extends outside the process's allocated address space.
ELOOP	Too many symbolic links were encountered in translating *path*.
EINVAL	The named file is not a symbolic link.
EIO	An I/O error occurred while reading from or writing to the file system.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} (see **sysconf**(2V)) while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	The named file does not exist.

SEE ALSO

stat(2V), **symlink**(2)

NAME

reboot – reboot system or halt processor

SYNOPSIS

#include <sys/reboot.h>

reboot(howto, [bootargs])
int howto;
char *bootargs;

DESCRIPTION

reboot( ) reboots the system, and is invoked automatically in the event of unrecoverable system failures. *howto* is a mask of options passed to the bootstrap program. The system call interface permits only **RB_HALT** or **RB_AUTOBOOT** to be passed to the reboot program; the other flags are used in scripts stored on the console storage media, or used in manual bootstrap procedures. When none of these options (for instance **RB_AUTOBOOT**) is given, the system is rebooted from file **/vmunix** in the root file system of unit 0 of a disk chosen in a processor specific way. An automatic consistency check of the disks is then normally performed.

The bits of *howto* are:

RB_HALT the processor is simply halted; no reboot takes place. **RB_HALT** should be used with caution.

RB_ASKNAME Interpreted by the bootstrap program itself, causing it to inquire as to what file should be booted. Normally, the system is booted from the file **/vmunix** without asking.

RB_SINGLE Normally, the reboot procedure involves an automatic disk consistency check and then multi-user operations. **RB_SINGLE** prevents the consistency check, rather simply booting the system with a single-user shell on the console. **RB_SINGLE** is interpreted by the **init**(8) program in the newly booted system.

RB_DUMP A system core dump is performed before rebooting.

RB_STRING The optional argument *bootargs* is passed to the bootstrap program. See **boot**(8S) for details. This option overrides **RB_SINGLE** but the same effect can be achieved by including **–s** as an option in *bootargs*.

Only the super-user may **reboot( )** a machine.

RETURN VALUES

On success, **reboot( )** does not return. On failure, it returns –1 and sets **errno** to indicate the error.

ERRORS

EPERM The caller is not the super-user.

FILES

/vmunix

SEE ALSO

panic(8S), **halt**(8), **init**(8), **intro**(8), **reboot**(8)

NAME

recv, recvfrom, recvmsg – receive a message from a socket

SYNOPSIS

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

int recv(s, buf, len, flags)
int s;
char *buf;
int len, flags;

int recvfrom(s, buf, len, flags, from, fromlen)
int s;
char *buf;
int len, flags;
struct sockaddr *from;
int *fromlen;

int recvmsg(s, msg, flags)
int s;
struct msghdr *msg;
int flags;
```

DESCRIPTION

s is a socket created with **socket**(2). **recv**( ), **recvfrom**( ), and **recvmsg**( ) are used to receive messages from another socket. **recv**( ) may be used only on a *connected* socket (see **connect**(2)), while **recvfrom**( ) and **recvmsg**( ) may be used to receive data on a socket whether it is in a connected state or not.

If *from* is not a NULL pointer, the source address of the message is filled in. *fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there. The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see **socket**(2)).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking (see **ioctl**(2)) in which case −1 is returned with the external variable **errno** set to EWOULDBLOCK.

The **select**(2) call may be used to determine when more data arrive.

If the process calling **recv**( ), **recvfrom**( ) or **recvmsg**( ) receives a signal before any data are available, the system call is restarted unless the calling process explicitly set the signal to interrupt these calls using **sigvec**( ) or **sigaction**( ) (see the discussions of **SV_INTERRUPT** on **sigvec**(2), and **SA_INTERRUPT** on **sigaction**(3V)).

The *flags* parameter is formed by ORing one or more of the following:

MSG_OOB Read any "out-of-band" data present on the socket, rather than the regular "in-band" data.

MSG_PEEK "Peek" at the data present on the socket; the data are returned, but not consumed, so that a subsequent receive operation will see the same data.

The **recvmsg()** call uses a **msghdr** structure to minimize the number of directly supplied parameters. This structure is defined in **<sys/socket.h>**, and includes the following members:

```
caddr_t     msg_name;       /* optional address */
int         msg_namelen;    /* size of address */
struct iovec *msg_iov;      /* scatter/gather array */
int         msg_iovlen;     /* # elements in msg_iov */
caddr_t     msg_accrights;  /* access rights sent/received */
int         msg_accrightslen;
```

Here **msg_name** and **msg_namelen** specify the destination address if the socket is unconnected; **msg_name** may be given as a NULL pointer if no names are desired or required. The **msg_iov** and **msg_iovlen** describe the scatter-gather locations, as described in **read**(2V). A buffer to receive any access rights sent along with the message is specified in **msg_accrights**, which has length **msg_accrightslen**.

RETURN VALUES

These calls return the number of bytes received, or −1 if an error occurred.

ERRORS

EBADF	*s* is an invalid descriptor.
EFAULT	The data were specified to be received into a non-existent or protected part of the process address space.
EINTR	The calling process received a signal before any data were available to be received, and the signal was set to interrupt the system call.
ENOTSOCK	*s* is a descriptor for a file, not a socket.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.

SEE ALSO

connect(2), **fcntl**(2V), **getsockopt**(2), **ioctl**(2), **read**(2V), **select**(2), **send**(2), **socket**(2)

NAME

rename – change the name of a file

SYNOPSIS

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

DESCRIPTION

rename( ) renames the link named *path1* as *path2*. If *path2* exists, then it is first removed. If *path2* refers to a directory, it must be an empty directory, and must not include *path1* in its path prefix. Both *path1* and *path2* must be of the same type (that is, both directories or both non-directories), and must reside on the same file system. Write access permission is required for both the directory containing *path1* and the directory containing *path2*. If a rename request relocates a directory in the hierarchy, write permission in the directory to be moved is needed, since its entry for the parent directory (..) must be updated.

rename( ) guarantees that an instance of *path2* will always exist, even if the system should crash in the middle of the operation.

If the final component of *path1* is a symbolic link, the symbolic link is renamed, not the file or directory to which it points.

If the file referred to by *path2* exists and the file's link count becomes zero when it is removed and no process has the file open, the space occupied by the file is freed, and the file is no longer accessible. If one or more processes have the file open when the last link is removed, the link is removed before **rename( )** returns, but the file's contents are not removed until all references to the file have been closed.

Upon successful completion, **rename( )** marks for update the **st_ctime** and **st_mtime** fields of the parent directory of each file.

RETURN VALUES

rename( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

rename( ) will fail and neither *path1* nor *path2* will be affected if:

EACCES	Write access is denied for either *path1* or *path2*.
	A component of the path prefix of either *path1* or *path2* denies search permission.
	The requested rename requires writing in a directory with access permissions that deny write permission.
EBUSY	*path2* is a directory and is the mount point for a mounted file system.
EDQUOT	The directory in which the entry for the new name is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
EFAULT	Either or both of *path1* or *path2* point outside the process's allocated address space.
EINVAL	*path1* is a parent directory of *path2*.
	An attempt was made to rename '.' or '..'.
EIO	An I/O error occurred while reading from or writing to the file system.
EISDIR	*path2* points to a directory and *path1* points to a file that is not a directory.
ELOOP	Too many symbolic links were encountered while translating either *path1* or *path2*.

ENAMETOOLONG	The length of either path argument exceeds {PATH_MAX}. A pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	A component of the path prefix of either *path1* or *path2* does not exist. The file named by *path1* does not exist.
ENOSPC	The directory in which the entry for the new name is being placed cannot be extended because there is no space left on the file system containing the directory.
ENOTDIR	A component of the path prefix of either *path1* or *path2* is not a directory. *path1* names a directory and *path2* names a nondirectory file.
ENOTEMPTY	*path2* is a directory and is not empty.
EROFS	The requested rename requires writing in a directory on a read-only file system.
EXDEV	The link named by *path2* and the file named by *path1* are on different logical devices (file systems).

SYSTEM V ERRORS

In addition to the above, the following may also occur:

ENOENT	*path1* or *path2* points to an empty string.

SEE ALSO

open(2V)

WARNINGS

The system can deadlock if a loop in the file system graph is present. This loop takes the form of an entry in directory **a**, say **a/file1**, being a hard link to directory **b**, and an entry in directory **b**, say **b/file2**, being a hard link to directory **a**. When such a loop exists and two separate processes attempt to perform '**rename a/file1 b/file2**' and '**rename b/file2 a/file1**', respectively, the system may deadlock attempting to lock both directories for modification. Hard links to directories should not be used. System administrators should use symbolic links instead.

NAME

rmdir – remove a directory file

SYNOPSIS

int rmdir(path)
char *path;

DESCRIPTION

rmdir( ) removes a directory file whose name is given by *path*. The directory must not have any entries other than '.' and '..'. The directory must not be the root directory or the current directory of the calling process.

If the directory's link count becomes zero, and no process has the directory open, the space occupied by the directory is freed and the directory is no longer accessible. If one or more processes have the directory open when the last link is removed, the '.' and '..'. entries, if present, are removed before **rmdir( )** returns and no new entries may be created in the directory, but the directory is not removed until all references to the directory have been closed.

Upon successful completion, **rmdir( )** marks for update the **st_ctime** and **st_mtime** fields of the parent directory.

RETURN VALUES

rmdir( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

EACCES	Search permission is denied for a component of the path prefix of *path*.
EACCES	Write permission is denied for the parent directory of the directory to be removed.
EBUSY	The directory to be removed is the mount point for a mounted file system, or is being used by another process.
EFAULT	*path* points outside the process's allocated address space.
EINVAL	The directory referred to by *path* is the current directory, '.'.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	The directory referred to by *path* does not exist.
ENOTDIR	A component of the path prefix of *path* is not a directory.
ENOTDIR	The file referred to by *path* is not a directory.
ENOTEMPTY	The directory referred to by *path* contains files other than '.' and '..'.
EROFS	The directory to be removed resides on a read-only file system.

SYSTEM V ERRORS

In addition to the above, the following may also occur:

ENOENT	*path* points to a null pathname.

SEE ALSO

mkdir(2V), **unlink**(2V)

NAME

select – synchronous I/O multiplexing

SYNOPSIS

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

int select (width, readfds, writefds, exceptfds, timeout)
int width;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;

FD_SET (fd, &fdset)
FD_CLR (fd, &fdset)
FD_ISSET (fd, &fdset)
FD_ZERO (&fdset)
int fd;
fd_set fdset;
```

DESCRIPTION

select( ) examines the I/O descriptor sets whose addresses are passed in *readfds*, *writefds*, and *exceptfds* to see if some of their descriptors are ready for reading, ready for writing, or have an exceptional condition pending. *width* is the number of bits to be checked in each bit mask that represent a file descriptor; the descriptors from 0 through *width*–1 in the descriptor sets are examined. Typically *width* has the value returned by **ulimit**(3C) for the maximum number of file descriptors. On return, **select( )** replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: **FD_ZERO** (*&fdset*) initializes a descriptor set *fdset* to the null set. **FD_SET**(*fd, &fdset*) includes a particular descriptor *fd* in *fdset*. **FD_CLR**(*fd, &fdset*) removes *fd* from *fdset*. **FD_ISSET**(*fd, &fdset*) is nonzero if *fd* is a member of *fdset*, zero otherwise. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to **FD_SETSIZE**, which is normally at least equal to the maximum number of descriptors supported by the system.

If *timeout* is not a NULL pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a NULL pointer, the select blocks indefinitely. To effect a poll, the *timeout* argument should be a non-NULL pointer, pointing to a zero-valued **timeval** structure.

Any of *readfds*, *writefds*, and *exceptfds* may be given as NULL pointers if no descriptors are of interest.

Selecting true for reading on a socket descriptor upon which a **listen**(2) call has been performed indicates that a subsequent **accept**(2) call on that descriptor will not block.

RETURN VALUES

select( ) returns a non-negative value on success. A positive value indicates the number of ready descriptors in the descriptor sets. 0 indicates that the time limit referred to by *timeout* expired. On failure, **select( )** returns –1, sets **errno** to indicate the error, and the descriptor sets are not changed.

ERRORS

EBADF	One of the descriptor sets specified an invalid descriptor.
EFAULT	One of the pointers given in the call referred to a non-existent portion of the process' address space.
EINTR	A signal was delivered before any of the selected events occurred, or before the time limit expired.
EINVAL	A component of the pointed-to time limit is outside the acceptable range: **t_sec** must be between 0 and 10^8, inclusive. **t_usec** must be greater than or equal to 0, and less than 10^6.

SEE ALSO

accept(2), **connect**(2), **fcntl**(2V), **ulimit**(3C), **gettimeofday**(2), **listen**(2), **read**(2V), **recv**(2), **send**(2), **write**(2V)

NOTES

Under rare circumstances, **select**( ) may indicate that a descriptor is ready for writing when in fact an attempt to write would block. This can happen if system resources necessary for a write are exhausted or otherwise unavailable. If an application deems it critical that writes to a file descriptor not block, it should set the descriptor for non-blocking I/O using the **F_SETFL** request to **fcntl**(2V).

BUGS

Although the provision of **ulimit**(3C) was intended to allow user programs to be written independent of the kernel limit on the number of open files, the dimension of a sufficiently large bit field for select remains a problem. The default size **FD_SETSIZE** (currently 256) is somewhat larger than the current kernel limit to the number of open files. However, in order to accommodate programs which might potentially use a larger number of open files with select, it is possible to increase this size within a program by providing a larger definition of **FD_SETSIZE** before the inclusion of **<sys/types.h>**.

select( ) should probably return the time remaining from the original timeout, if any, by modifying the time value in place. This may be implemented in future versions of the system. Thus, it is unwise to assume that the timeout pointer will be unmodified by the **select**( ) call.

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, semnum, cmd;
union semun {
        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 the array pointed to by *arg.array*. **[READ]**

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

The following *cmd*s are also available:

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

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

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

This *cmd* can only be executed by a process that has an effective user ID equal to either that of super-user, or to the value of **sem_perm.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*.

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

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

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

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

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

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

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

Otherwise, the corresponding permissions are denied.

RETURN VALUES

On success, the value returned by **semctl**( ) 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	0.

On failure, **semctl**( ) returns −1 and sets **errno** to indicate the error.

ERRORS

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

SEE ALSO

intro(2), **semget**(2), **semop**(2), **ipcrm**(1), **ipcs**(1)

NAME

semget – get set of semaphores

SYNOPSIS

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

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

DESCRIPTION

semget() returns the semaphore identifier associated with *key*.

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

- *key* is equal to **IPC_PRIVATE**.
- *key* does not already have a semaphore identifier associated with it, and (*semflg* & **IPC_CREAT**) is "true".

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

- **sem_perm.cuid**, **sem_perm.uid**, **sem_perm.cgid**, and **sem_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of **sem_perm.mode** are set equal to the low-order 9 bits of *semflg*.
- **sem_nsems** is set equal to the value of *nsems*.
- **sem_otime** is set equal to 0 and **sem_ctime** is set equal to the current time.

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

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

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

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

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

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

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

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

RETURN VALUES

semget( ) returns a non-negative semaphore identifier on success. On failure, it returns −1 and sets **errno** to indicate the error.

ERRORS

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.

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

EINVAL — *nsems* is either less than or equal to zero or greater than the system-imposed limit.

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.

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

SEE ALSO

ipcrm(1), **ipcs**(1), **intro**(2), **semctl**(2), **semop**(2)

NAME

semop – semaphore operations

SYNOPSIS

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

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

DESCRIPTION

semop( ) is used to perform atomically 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] (see **semctl**(2))

- 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**(2V)) 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**(3V).

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

If **sem_op( )** is zero, one of the following will occur: **[READ]**

- If *semval* is zero, **semop( )** will return immediately.
- If *semval* is not equal to zero and (**sem_flg** & **IPC_NOWAIT**) is "true", **semop( )** will return immediately.
- If *semval* is not equal to zero and (**sem_flg** & **IPC_NOWAIT**) is "false", **semop( )** will increment the *semzcnt* associated with the specified semaphore and suspend execution of the calling process until one of the following occurs:
    - *semval* becomes zero, at which time the value of *semzcnt* associated with the specified semaphore is decremented.
    - The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, **errno** is set equal to EIDRM, and a value of −1 is returned.
    - The calling process receives a signal that is to be caught. When this occurs, the value of *semzcnt* associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in **signal**(3V).

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

RETURN VALUES

semop( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

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".
EFAULT	*sops* points to an illegal address.
EFBIG	**sem_num** is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*.
EIDRM	The set of semaphores referred to by *msqid* was removed from the system.
EINTR	The call was interrupted by the delivery of a signal.
EINVAL	*semid* is not a valid semaphore identifier.
	The number of individual semaphores for which the calling process requests a **SEM_UNDO** would exceed the limit.
ENOSPC	The limit on the number of individual processes requesting an **SEM_UNDO** would be exceeded.
ERANGE	An operation would cause a *semval* or *semadj* value to overflow the system-imposed limit.

SEE ALSO

ipcrm(1), **ipcs**(1), **intro**(2), **execve**(2V), **exit**(2V), **fork**(2V), **semctl**(2), **semget**(2), **signal**(3V)

NAME

send, sendto, sendmsg – send a message from a socket

SYNOPSIS

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

int send(s, msg, len, flags)
int s;
char *msg;
int len, flags;

int sendto(s, msg, len, flags, to, tolen)
int s;
char *msg;
int len, flags;
struct sockaddr *to;
int tolen;

int sendmsg(s, msg, flags)
int s;
struct msghdr *msg;
int flags;
```

DESCRIPTION

s is a socket created with **socket**(2). **send**(), **sendto**(), and **sendmsg**() are used to transmit a message to another socket. **send**() may be used only when the socket is in a *connected* state, while **sendto**() and **sendmsg**() may be used at any time.

The address of the target is given by *to* with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, then the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a **send**(). Return values of −1 indicate some locally detected errors.

If no buffer space is available at the socket to hold the message to be transmitted, then **send**() normally blocks, unless the socket has been placed in non-blocking I/O mode. The **select**(2) call may be used to determine when it is possible to send more data.

If the process calling **send**(), **sendmsg**() or **sendto**() receives a signal before any data are buffered to be sent, the system call is restarted unless the calling process explicitly set the signal to interrupt these calls using **sigvec**() or **sigaction**() (see the discussions of SV_INTERRUPT on **sigvec**(2), and SA_INTERRUPT on **sigaction**(3V)).

The *flags* parameter is formed by ORing one or more of the following:

MSG_OOB	Send ''out-of-band'' data on sockets that support this notion. The underlying protocol must also support ''out-of-band'' data. Currently, only SOCK_STREAM sockets created in the AF_INET address family support out-of-band data.
MSG_DONTROUTE	The SO_DONTROUTE option is turned on for the duration of the operation. This is usually used only by diagnostic or routing programs.

See **recv**(2) for a description of the **msghdr** structure.

RETURN VALUES

On success, these functions return the number of bytes sent. On failure, they return −1 and set **errno** to indicate the error.

ERRORS

EBADF	*s* is an invalid descriptor.
EFAULT	The data was specified to be sent to a non-existent or protected part of the process address space.
EINTR	The calling process received a signal before any data could be buffered to be sent, and the signal was set to interrupt the system call.
EINVAL	*len* is not the size of a valid address for the specified address family.
EMSGSIZE	The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.
ENOBUFS	The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.
ENOBUFS	The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.
ENOTSOCK	*s* is a descriptor for a file, not a socket.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.

SEE ALSO

connect(2), **fcntl**(2V), **getsockopt**(2), **recv**(2), **select**(2), **socket**(2), **write**(2V)

NAME

setpgid – set process group ID for job control

SYNOPSIS

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

int setpgid (pid, pgid)
pid_t pid, pgid;
```

DESCRIPTION

setpgid( ) is used to either join an existing process group or create a new process group within the session of the calling process (see NOTES). The process group ID of a session leader does not change. Upon successful completion, the process group ID of the process with a process ID that matches *pid* is set to *pgid*. As a special case, if *pid* is zero, the process ID of the calling process is used. Also, if *pgid* is zero, the process ID of the process indicated by *pid* is used.

RETURN VALUES

setpgid( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

EACCES The value of *pid* matches the process ID of a child process of the calling process and the child process has successfully executed one of the **exec( )** functions.

EINVAL The value of *pgid* is less than zero or is greater than MAXPID, the maximum process ID as defined in **<sys/param.h>**.

EPERM The process indicated by *pid* is a session leader. The value of *pid* is valid but matches the process ID of a child process of the calling process and the child process is not in the same session as the calling process. The value of *pgid* does not match the process ID of the process indicated by *pid* and there is no process with a process group ID that matches the value of *pgid* in the same session as the calling process.

ESRCH *pid* does not match the PID of the calling process or the PID of a child of the calling process.

SEE ALSO

getpgrp(2V), **execve**(2V), **setsid**(2V), **tcgetpgrp**(3V)

NOTES

For **setpgid( )** to behave as described above, {_POSIX_JOB_CONTROL} must be in effect (see **sysconf**(2V)). {_POSIX_JOB_CONTROL} is always in effect on SunOS systems, but for portability, applications should call **sysconf( )** to determine whether {_POSIX_JOB_CONTROL} is in effect for the current system.

NAME

setregid – set real and effective group IDs

SYNOPSIS

int setregid(rgid, egid)
int rgid, egid;

DESCRIPTION

setregid( ) is used to set the real and effective group IDs of the calling process. If *rgid* is –1, the real GID is not changed; if *egid* is –1, the effective GID is not changed. The real and effective GIDs may be set to different values in the same call.

If the effective user ID of the calling process is super-user, the real GID and the effective GID can be set to any legal value.

If the effective user ID of the calling process is not super-user, either the real GID can be set to the saved setGID from **execve**(2V), or the effective GID can either be set to the saved setGID or the real GID. Note: if a setGID process sets its effective GID to its real GID, it can still set its effective GID back to the saved setGID.

In either case, if the real GID is being changed (that is, if *rgid* is not –1), or the effective GID is being changed to a value not equal to the real GID, the saved setGID is set equal to the new effective GID.

RETURN VALUES

setregid( ) returns:

0 on success.

–1 on failure and sets **errno** to indicate the error.

ERRORS

setregid( ) will fail and neither of the group IDs will be changed if:

EINVAL The value of *rgid* or *egid* is less than 0 or greater than USHRT_MAX (defined in **<sys/limits.h>**).

EPERM The calling process' effective UID is not the super-user and a change other than changing the real GID to the saved setGID, or changing the effective GID to the real GID or the saved GID, was specified.

SEE ALSO

execve(2V), **getgid**(2V), **setreuid**(2), **setuid**(3V)

NAME

setreuid – set real and effective user IDs

SYNOPSIS

int setreuid(ruid, euid)
int ruid, euid;

DESCRIPTION

setreuid( ) is used to set the real and effective user IDs of the calling process. If *ruid* is −1, the real user ID is not changed; if *euid* is −1, the effective user ID is not changed. The real and effective user IDs may be set to different values in the same call.

If the effective user ID of the calling process is super-user, the real user ID and the effective user ID can be set to any legal value.

If the effective user ID of the calling process is not super-user, either the real user ID can be set to the effective user ID, or the effective user ID can either be set to the saved set-user ID from **execve**(2V) or the real user ID. Note: if a set-UID process sets its effective user ID to its real user ID, it can still set its effective user ID back to the saved set-user ID.

In either case, if the real user ID is being changed (that is, if *ruid* is not −1), or the effective user ID is being changed to a value not equal to the real user ID, the saved set-user ID is set equal to the new effective user ID.

RETURN VALUES

setreuid( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

setreuid( ) will fail and neither of the user IDs will be changed if:

EINVAL The value of *ruid* or *euid* is less than 0 or greater than USHRT_MAX (defined in **<sys/limits.h>**).

EPERM The calling process' effective user ID is not the super-user and a change other than changing the real user ID to the effective user ID, or changing the effective user ID to the real user ID or the saved set-user ID, was specified.

SEE ALSO

execve(2V), **getuid**(2V), **setregid**(2), **setuid**(3V)

NAME

setsid – create session and set process group ID

SYNOPSIS

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

pid_t setsid()
```

DESCRIPTION

If the calling process is not a process group leader, the **setsid()** function creates a new session. The calling process is the session leader of this new session, the process group leader of a new process group, and has no controlling terminal. If the process had a controlling terminal, **setsid()** breaks the association between the process and that controlling terminal. The process group ID of the calling process is set equal to the process ID of the calling process. The calling process is the only process in the new process group and the only process in the new session.

RETURN VALUES

setsid() returns the process group ID of the calling process on success. On failure, it returns −1 and sets **errno** to indicate the error.

ERRORS

If any of the following conditions occur, **setsid()** returns −1 and sets **errno** to the corresponding value:

EPERM The calling process is already a process group leader.

The process ID of the calling process equals the process group ID of a different process.

SEE ALSO

execve(2V), **exit**(2V), **fork**(2V), **getpid**(2V), **getpgrp**(2V), **kill**(2V), **setpgid**(2V), **sigaction**(3V)

NAME

setuseraudit, setaudit – set the audit classes for a specified user ID

SYNOPSIS

```
#include <sys/label.h>
#include <sys/audit.h>

int setuseraudit(uid, state)
int uid;
audit_state_t *state;

int setaudit(state)
audit_state_t *state;
```

DESCRIPTION

The **setuseraudit( )** system call sets the audit state for all processes whose audit user ID matches the specified user ID. The parameter *state* specifies the audit classes to audit for both successful and unsuccessful operations.

The **setaudit( )** system call sets the audit state for the current process.

Only processes with the real or effective user ID of the super-user may successfully execute these calls.

RETURN VALUES

setuseraudit( ) and **setaudit( )** return:

0 on success.

–1 on failure and set **errno** to indicate the error.

ERRORS

EFAULT The *state* parameter points outside the processes' allocated address space.

EPERM The process' real or effective user ID is not super-user.

SEE ALSO

audit(2), **audit_args**(3), **audit_control**(5), **audit.log**(5)

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. If no processes are currently mapped to the corresponding shared memory segment, then the segment is removed and the associated resources are reclaimed. Otherwise, the segment will persist, although **shmget**(2) will not be able to locate it, until it is no longer mapped by any process. 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*.

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

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

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

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

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

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

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

Otherwise, the corresponding permissions are denied.

RETURN VALUES

shmctl( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

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

EFAULT *buf* points to an illegal address.

EINVAL *shmid* is not a valid shared memory identifier.

cmd is not a valid command.

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

SEE ALSO

ipcrm(1), **ipcs**(1), **intro**(2), **shmget**(2), **shmop**(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)
key_t key;
int size, shmflg;
```

DESCRIPTION

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

A shared memory identifier and associated data structure and shared memory segment of 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.

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

```
struct   ipc_perm shm_perm;  /* operation permission struct */
int      shm_segsz;          /* size of segment */
ushort   shm_cpid;           /* creator pid */
ushort   shm_lpid;           /* pid of last operation */
short    shm_nattch;         /* number of current attaches */
time_t   shm_atime;          /* last attach time */
time_t   shm_dtime;          /* last detach time */
time_t   shm_ctime;          /* last change time */
                             /* Times measured in secs since */
                             /* 00:00:00 GMT, Jan. 1, 1970 */
```

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

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

shm_segsz specifies the size of the shared memory segment. **shm_cpid** is the process ID of the process that created the shared memory identifier. **shm_lpid** is the process ID of the last process that performed a **shmop**(2) operation. **shm_nattch** is the number of processes that currently have this segment attached. **shm_atime** is the time of the last *shmat* operation, **shm_dtime** is the time of the last *shmdt* operation, and **shm_ctime** is the time of the last **shmctl**(2) operation that changed one of the members of the above structure.

RETURN VALUES

shmget( ) returns a non-negative shared memory identifier on success. On failure, it returns −1 and sets **errno** to indicate the error.

ERRORS

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.
EEXIST	A shared memory identifier exists for *key* but ((*shmflg* & **IPC_CREAT**) && (*shmflg* & **IPC_EXCL**)) is "true".
EINVAL	*size* is less than the system-imposed minimum or greater than the system-imposed maximum. 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".
ENOMEM	A shared memory identifier and associated shared memory segment are to be created but the amount of available physical memory is not sufficient to fill the request.
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.

SEE ALSO

ipcrm(1), **ipcs**(1), **intro**(2), **shmctl**(2), **shmop**(2)

NAME

shmop, shmat, shmdt – 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() maps the shared memory segment associated with the shared memory identifier specified by *shmid* into the data segment of the calling process. Upon successful completion, the address of the mapped segment is returned.

The shared memory segment is mapped at the address specified by one of the following criteria:

- If *shmaddr* is equal to zero, the segment is mapped at an address selected by the system. Ordinarily, applications should invoke **shmat()** with *shmaddr* equal to zero so that the operating system may make the best use of available resources.
- If *shmaddr* is not equal to zero and (*shmflg* & **SHM_RND**) is "true", the segment is mapped 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 mapped at the address given by *shmaddr*.

The segment is mapped for reading if (*shmflg* & **SHM_RDONLY**) is "true" **[READ]**, otherwise it is mapped for reading and writing **[READ/WRITE]** (see **shmctl**(2)).

shmdt() unmaps from the calling process's address space the shared memory segment that is mapped at the address specified by *shmaddr*. The shared memory segment must have been mapped with a prior **shmat()** function call. The segment and contents are retained until explicitly removed by means of the **IPC_RMID** function (see **shmctl**(2)).

RETURN VALUES

shmat() returns the data segment start address of the mapped shared memory segment. On failure, it returns –1 and sets **errno** to indicate the error.

shmdt() returns:

0	on success.
–1	on failure and sets **errno** to indicate the error.

ERRORS

shmat() will fail and not map the shared memory segment if one or more of the following are true:

EACCES	Operation permission is denied to the calling process (see **intro**(2)).
EINVAL	*shmid* is not a valid shared memory identifier. *shmaddr* is not equal to zero, and the value of (*shmaddr* - (*shmaddr* modulus **SHMLBA**)) is an illegal address. *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 mapped to the calling process would exceed the system-imposed limit.

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

shmdt() will fail and not unmap the shared memory segment if:

EINVAL *shmaddr* is not the data segment start address of a shared memory segment.

SEE ALSO

ipcrm(1), **ipcs**(1), **intro**(2), **execve**(2V), **exit**(2V), **fork**(2V), **shmctl**(2), **shmget**(2)

NAME

shutdown – shut down part of a full-duplex connection

SYNOPSIS

int shutdown(s, how)
int s, how;

DESCRIPTION

The **shutdown()** call causes all or part of a full-duplex connection on the socket associated with *s* to be shut down. If *how* is 0, then further receives will be disallowed. If *how* is 1, then further sends will be disallowed. If *how* is 2, then further sends and receives will be disallowed.

RETURN VALUES

shutdown() returns:

0	on success.
−1	on failure and sets **errno** to indicate the error.

ERRORS

EBADF	*s* is not a valid descriptor.
ENOTCONN	The specified socket is not connected.
ENOTSOCK	*s* is a file, not a socket.

SEE ALSO

ipcrm(1), **ipcs**(1), **connect**(2), **socket**(2)

BUGS

The *how* values should be defined constants.

NAME

sigblock, sigmask – block signals

SYNOPSIS

```
#include <signal.h>

int sigblock(mask);
int mask;

int sigmask(signum)
```

DESCRIPTION

sigblock() adds the signals specified in *mask* to the set of signals currently being blocked from delivery. A signal is blocked if the appropriate bit in *mask* is set. The macro **sigmask()** is provided to construct the signal mask for a given *signum*. **sigblock()** returns the previous signal mask, which may be restored using **sigsetmask**(2).

It is not possible to block SIGKILL or SIGSTOP. The system silently imposes this restriction.

RETURN VALUES

sigblock() returns the previous signal mask.

The **sigmask()** macro returns the mask for the given signal number.

SEE ALSO

kill(2V), **sigsetmask**(2), **sigvec**(2), **signal**(3V)

NAME

sigpause, sigsuspend – automatically release blocked signals and wait for interrupt

SYNOPSIS

int sigpause(sigmask)
int sigmask;

#include <signal.h>

int sigsuspend(sigmaskp)
sigset_t *sigmaskp;

DESCRIPTION

sigpause() assigns *sigmask* to the set of masked signals and then waits for a signal to arrive; on return the set of masked signals is restored. *sigmask* is usually 0 to indicate that no signals are now to be blocked. **sigpause()** always terminates by being interrupted, returning EINTR.

In normal usage, a signal is blocked using **sigblock**(2), to begin a critical section, variables modified on the occurrence of the signal are examined to determine that there is no work to be done, and the process pauses awaiting work by using **sigpause()** with the mask returned by **sigblock()**.

sigsuspend() replaces the process's signal mask with the set of signals pointed to by *sigmaskp* and then suspends the process until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. If the action is to terminate the process, **sigsuspend()** does not return. If the action is to execute a signal-catching function, **sigsuspend()** returns after the signal-catching function returns, with the signal mask restored to the setting that existed prior to the **sigsuspend()** call. It is not possible to block those signals that cannot be ignored, as documented in **<signal.h>** this is enforced by the system without indicating an error.

RETURN VALUES

Since **sigpause()** and **sigsuspend()** suspend process execution indefinitely, there is no successful completion return value. On failure, these functions return –1 and set **errno** to indicate the error.

ERRORS

EINTR	A signal is caught by the calling process and control is returned from the signal-catching function.

SEE ALSO

sigblock(2), **sigpending**(2V), **sigprocmask**(2V), **sigvec**(2), **pause**(3V), **sigaction**(3V), **signal**(3V), **sigsetops**(3V)

NAME

sigpending – examine pending signals

SYNOPSIS

```
#include <signal.h>

int sigpending(set)
sigset_t *set;
```

DESCRIPTION

sigpending( ) stores the set of signals that are blocked from delivery and pending for the calling process in the space pointed to by *set*.

RETURN VALUES

sigpending( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

SEE ALSO

sigprocmask(2V), **sigvec**(2), **sigsetops**(3V)

NAME

sigprocmask – examine and change blocked signals

SYNOPSIS

```
#include <signal.h>

int sigprocmask(how, set, oset)
int how;
sigset_t *set, *oset;
```

DESCRIPTION

sigprocmask() is used to examine or change (or both) the calling process's signal mask. If the value of *set* is not NULL, it points to a set of signals to be used to change the currently blocked set.

The value of *how* indicates the manner in which the set is changed, and consists of one of the following values, as defined in the header **<signal.h>**:

SIG_BLOCK	The resulting set is the union of the current set and the signal set pointed to by *set*.
SIG_UNBLOCK	The resulting set is the intersection of the current set and the complement of the signal set pointed to by *set*.
SIG_SETMASK	The resulting set is the signal set pointed to by *set*.

If *oset* is not NULL, the previous mask is stored in the space pointed to by *oset*. If the value of *set* is NULL, the value of *how* is not significant and the process's signal mask is unchanged by this function call. Thus, the call can be used to enquire about currently blocked signals.

If there are any pending unblocked signals after the call to **sigprocmask()**, at least one of those signals is be delivered before **sigprocmask()** returns.

If it is not possible to block the **SIGKILL** and **SIGSTOP** signals. This is enforced by the system without causing an error to be indicated.

If any of the **SIGFPE**, **SIGKILL**, or **SIGSEGV** signals are generated while they are blocked, the result is undefined, unless the signal was generated by a call to **kill**(2V).

If **sigprocmask()** fails, the process's signal mask is not changed.

RETURN VALUES

sigprocmask() returns:

0	on success.
−1	on failure and sets **errno** to indicate the error.

ERRORS

EINVAL	The value of *how* is not equal to one of the defined values.

SEE ALSO

sigpause(2V), **sigpending**(2V), **sigvec**(2), **sigaction**(3V), **sigsetops**(3V)

NAME

sigsetmask – set current signal mask

SYNOPSIS

```
#include <signal.h>

int sigsetmask(mask)
int mask;
```

DESCRIPTION

sigsetmask() sets the set of signals currently being blocked from delivery according to *mask*. A signal is blocked if the appropriate bit in *mask* is set. The macro **sigblock**(2) is provided to construct the mask for a given *signum*.

The system silently disallows blocking **SIGKILL** and **SIGSTOP**.

RETURN VALUES

sigsetmask() returns the previous signal mask.

SEE ALSO

kill(2V), **sigblock**(2), **sigpause**(2V), **sigvec**(2), **signal**(3V)

NAME

sigstack – set and/or get signal stack context

SYNOPSIS

```
#include <signal.h>

int sigstack (ss, oss)
struct sigstack *ss, *oss;
```

DESCRIPTION

sigstack( ) allows users to define an alternate stack, called the "signal stack", on which signals are to be processed. When a signal's action indicates its handler should execute on the signal stack (specified with a **sigvec**(2) call), the system checks to see if the process is currently executing on that stack. If the process is not currently executing on the signal stack, the system arranges a switch to the signal stack for the duration of the signal handler's execution.

A signal stack is specified by a **sigstack( )** structure, which includes the following members:

```
char    *ss_sp;       /* signal stack pointer */
int     ss_onstack;   /* current status */
```

ss_sp is the initial value to be assigned to the stack pointer when the system switches the process to the signal stack. Note that, on machines where the stack grows downwards in memory, this is *not* the address of the beginning of the signal stack area. **ss_onstack** field is zero or non-zero depending on whether the process is currently executing on the signal stack or not.

If *ss* is not a NULL pointer, **sigstack( )** sets the signal stack state to the value in the **sigstack( )** structure pointed to by *ss*. Note: if **ss_onstack** is non-zero, the system will think that the process is executing on the signal stack. If *ss* is a NULL pointer, the signal stack state will be unchanged. If *oss* is not a NULL pointer, the current signal stack state is stored in the **sigstack( )** structure pointed to by *oss*.

RETURN VALUES

sigstack( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

sigstack( ) will fail and the signal stack context will remain unchanged if one of the following occurs.

EFAULT *ss* or *oss* points to memory that is not a valid part of the process address space.

SEE ALSO

sigvec(2), **setjmp**(3V), **signal**(3V)

NOTES

Signal stacks are not "grown" automatically, as is done for the normal stack. If the stack overflows unpredictable results may occur.

NAME

sigvec – software signal facilities

SYNOPSIS

```
#include <signal.h>

int sigvec(sig, vec, ovec)
int sig;
struct sigvec *vec, *ovec;
```

DESCRIPTION

The system defines a set of signals that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify a *handler* to which a signal is delivered, or specify that a signal is to be *blocked* or *ignored*. A process may also specify that a default action is to be taken by the system when a signal occurs. Normally, signal handlers execute on the current stack of the process. This may be changed, on a per-handler basis, so that signals are taken on a special *signal stack*.

All signals have the same *priority*. Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. A global *signal mask* defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent (normally 0). It may be changed with a **sigblock**(2) or **sigsetmask**(2) call, or when a signal is delivered to the process.

A process may also specify a set of *flags* for a signal that affect the delivery of that signal.

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process then it is delivered to the process. When a signal is delivered, the current state of the process is saved, a new signal mask is calculated (as described below), and the signal handler is invoked. The call to the handler is arranged so that if the signal handling routine returns normally the process will resume execution in the context from before the signal's delivery. If the process wishes to resume in a different context, then it must arrange to restore the previous context itself.

When a signal is delivered to a process a new signal mask is installed for the duration of the process' signal handler (or until a **sigblock**( ) or **sigsetmask**( ) call is made). This mask is formed by taking the current signal mask, adding the signal to be delivered, and ORing in the signal mask associated with the handler to be invoked.

The action to be taken when the signal is delivered is specified by a **sigvec** structure, defined in **<signal.h>** as:

```
struct sigvec {
        void (*sv_handler)();       /* signal handler */
        int sv_mask;                /* signal mask to apply */
        int sv_flags;               /* see signal options */
}
```

The following bits may be set in **sv_flags**:

```
#define SV_ONSTACK      0x0001      /* take signal on signal stack */
#define SV_INTERRUPT    0x0002      /* do not restart system on signal return */
#define SV_RESETHAND    0x0004      /* reset signal handler to SIG_DFL on signal */
```

If the SV_ONSTACK bit is set in the flags for that signal, the system will deliver the signal to the process on the signal stack specified with **sigstack**(2), rather than delivering the signal on the current stack.

If *vec* is not a NULL pointer, **sigvec**( ) assigns the handler specified by **sv_handler**, the mask specified by **sv_mask**, and the flags specified by **sv_flags** to the specified signal. If *vec* is a NULL pointer, **sigvec**( ) does not change the handler, mask, or flags for the specified signal.

The mask specified in *vec* is not allowed to block SIGKILL or SIGSTOP. The system enforces this restriction silently.

If *ovec* is not a NULL pointer, the handler, mask, and flags in effect for the signal before the call to **sigvec()** are returned to the user. A call to **sigvec()** with *vec* a NULL pointer and *ovec* not a NULL pointer can be used to determine the handling information currently in effect for a signal without changing that information.

The following is a list of all signals with names as in the include file **<signal.h>**:

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGILL	4*	illegal instruction
SIGTRAP	5*	trace trap
SIGABRT	6*	abort (generated by **abort**(3) routine)
SIGEMT	7*	emulator trap
SIGFPE	8*	arithmetic exception
SIGKILL	9	kill (cannot be caught, blocked, or ignored)
SIGBUS	10*	bus error
SIGSEGV	11*	segmentation violation
SIGSYS	12*	bad argument to system call
SIGPIPE	13	write on a pipe or other socket with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGURG	16•	urgent condition present on socket
SIGSTOP	17†	stop (cannot be caught, blocked, or ignored)
SIGTSTP	18†	stop signal generated from keyboard
SIGCONT	19•	continue after stop
SIGCHLD	20•	child status has changed
SIGTTIN	21†	background read attempted from control terminal
SIGTTOU	22†	background write attempted to control terminal
SIGIO	23•	I/O is possible on a descriptor (see **fcntl**(2V))
SIGXCPU	24	cpu time limit exceeded (see **getrlimit**(2))
SIGXFSZ	25	file size limit exceeded (see **getrlimit**(2))
SIGVTALRM	26	virtual time alarm (see **getitimer**(2))
SIGPROF	27	profiling timer alarm (see **getitimer**(2))
SIGWINCH	28•	window changed (see **termio**(4) and **win**(4S))
SIGLOST	29*	resource lost (see **lockd**(8C))
SIGUSR1	30	user-defined signal 1
SIGUSR2	31	user-defined signal 2

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

Once a signal handler is installed, it remains installed until another **sigvec()** call is made, or an **execve**(2V) is performed, unless the **SV_RESETHAND** bit is set in the flags for that signal. In that case, the value of the handler for the caught signal is set to **SIG_DFL** before entering the signal-catching function, unless the signal is **SIGILL** or **SIGTRAP**. Also, if this bit is set, the bit for that signal in the signal mask will not be set; unless the signal mask associated with that signal blocks that signal, further occurrences of that signal will not be blocked. The **SV_RESETHAND** flag is not available in 4.2BSD, hence it should not be used if backward compatibility is needed.

The default action for a signal may be reinstated by setting the signal's handler to **SIG_DFL**; this default is termination except for signals marked with • or †. Signals marked with • are discarded if the action is **SIG_DFL**; signals marked with † cause the process to stop. If the process is terminated, a "core image" will be made in the current working directory of the receiving process if the signal is one for which an asterisk appears in the above list *and* the following conditions are met:

- The effective user ID (EUID) and the real user ID (UID) of the receiving process are equal.
- The effective group ID (EGID) and the real group ID (GID) 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**(2V))
  - 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 file group ID of the current directory

If the handler for that signal is SIG_IGN, the signal is subsequently ignored, and pending instances of the signal are discarded.

Note: the signals SIGKILL and SIGSTOP cannot be ignored.

If a caught signal occurs during certain system calls, the call is restarted by default. The call can be forced to terminate prematurely with an EINTR error return by setting the SV_INTERRUPT bit in the flags for that signal. SV_INTERRUPT is not available in 4.2BSD, hence it should not be used if backward compatibility is needed. The affected system calls are **read**(2V) or **write**(2V) on a slow device (such as a terminal or pipe or other socket, but not a file) and during a **wait**(2V).

After a **fork**(2V), or **vfork**(2) the child inherits all signals, the signal mask, the signal stack, and the restart/interrupt and reset-signal-handler flags.

The **execve**(2V), call resets all caught signals to default action and resets all signals to be caught on the user stack. Ignored signals remain ignored; the signal mask remains the same; signals that interrupt system calls continue to do so.

CODES

The following defines the codes for signals which produce them. All of these symbols are defined in **signal.h**:

Condition	Signal	Code
Sun codes:		
Illegal instruction	**SIGILL**	**ILL_INSTR_FAULT**
Integer division by zero	**SIGFPE**	**FPE_INTDIV_TRAP**
IEEE floating pt inexact	**SIGFPE**	**FPE_FLTINEX_TRAP**
IEEE floating pt division by zero	**SIGFPE**	**FPE_FLTDIV_TRAP**
IEEE floating pt underflow	**SIGFPE**	**FPE_FLTUND_TRAP**
IEEE floating pt operand error	**SIGFPE**	**FPE_FLTOPERR_TRAP**
IEEE floating pt overflow	**SIGFPE**	**FPE_FLTOVF_FAULT**
Hardware bus error	**SIGBUS**	**BUS_HWERR**
Address alignment error	**SIGBUS**	**BUS_ALIGN**
No mapping fault	**SIGSEGV**	**SEGV_NOMAP**
Protection fault	**SIGSEGV**	**SEGV_PROT**
Object error	**SIGSEGV**	**SEGV_CODE(code)=SEGV_OBJERR**
Object error number	**SIGSEGV**	**SEGV_ERRNO(code)**
SPARC codes:		
Privileged instruction violation	**SIGILL**	**ILL_PRIVINSTR_FAULT**
Bad stack	**SIGILL**	**ILL_STACK**
Trap #*n* (1 <= *n* <= 127)	**SIGILL**	**ILL_TRAP_FAULT(*n*)**
Integer overflow	**SIGFPE**	**FPE_INTOVF_TRAP**
Tag overflow	**SIGEMT**	**EMT_TAG**
MC680X0 codes:		
Privilege violation	**SIGILL**	**ILL_PRIVVIO_FAULT**
Coprocessor protocol error	**SIGILL**	**ILL_INSTR_FAULT**
Trap #*n* (1 <= *n* <= 14)	**SIGILL**	**ILL_TRAP*n*_FAULT**
A-line op code	**SIGEMT**	**EMT_EMU1010**
F-line op code	**SIGEMT**	**EMT_EMU1111**
CHK or CHK2 instruction	**SIGFPE**	**FPE_CHKINST_TRAP**
TRAPV or TRAPcc or cpTRAPcc	**SIGFPE**	**FPE_TRAPV_TRAP**

IEEE floating pt compare unordered	SIGFPE	FPE_FLTBSUN_TRAP
IEEE floating pt signaling NaN	SIGFPE	FPE_FLTNAN_TRAP

ADDR

The *addr* signal handler parameter is defined as follows:

Signal	Code	Addr
Sun:		
SIGILL	Any	address of faulted instruction
SIGEMT	Any	address of faulted instruction
SIGFPE	Any	address of faulted instruction
SIGBUS	BUS_HWERR	address that caused fault
SIGSEGV	Any	address that caused fault
SPARC:		
SIGBUS	BUS_ALIGN	address of faulted instruction
MC680X0:		
SIGBUS	BUS_ALIGN	address that caused fault

The accuracy of *addr* is machine dependent. For example, certain machines may supply an address that is on the same page as the address that caused the fault. If an appropriate *addr* cannot be computed it will be set to SIG_NOADDR.

RETURN VALUES

sigvec( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

sigvec( ) will fail and no new signal handler will be installed if one of the following occurs:

EFAULT Either *vec* or *ovec* is not a NULL pointer and points to memory that is not a valid part of the process address space.

EINVAL *Sig* is not a valid signal number.

An attempt was made to ignore or supply a handler for SIGKILL or SIGSTOP.

SEE ALSO

execve(2V), **fcntl**(2V), **fork**(2V), **getitimer**(2), **getrlimit**(2), **ioctl**(2), **kill**(2V), **ptrace**(2), **read**(2V), **sigblock**(2), **sigpause**(2V), **sigsetmask**(2), **sigstack**(2), **umask**(2V), **vfork**(2), **wait**(2V), **write**(2V), **setjmp**(3V), **signal**(3V), **streamio**(4), **termio**(4), **win**(4S), **lockd**(8C)

NOTES

SIGPOLL is a synonym for SIGIO. A SIGIO will be issued when a file descriptor corresponding to a STREAMS (see **intro**(2)) file has a "selectable" event pending. Unless that descriptor has been put into asynchronous mode (see **fcntl (2V)**, a process must specifically request that this signal be sent using the I_SETSIG **ioctl**(2) call (see **streamio**(4)). Otherwise, the process will never receive SIGPOLL.

The handler routine can be declared:

```
void handler(sig, code, scp, addr)
int sig, code;
struct sigcontext *scp;
char *addr;
```

Here *sig* is the signal number; *code* is a parameter of certain signals that provides additional detail; *scp* is a pointer to the **sigcontext** structure (defined in **signal.h**), used to restore the context from before the signal; and *addr* is additional address information.

Programs that must be portable to UNIX systems other than 4.2BSD should use the **signal**(3V), interface instead.

NAME

socket – create an endpoint for communication

SYNOPSIS

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

int socket(domain, type, protocol)
int domain, type, protocol;

DESCRIPTION

socket( ) creates an endpoint for communication and returns a descriptor.

The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file **<sys/socket.h>**. The currently understood formats are

PF_UNIX	(UNIX system internal protocols),
PF_INET	(ARPA Internet protocols), and
PF_IMPLINK	(IMP "host at IMP" link layer).

The socket has the indicated *type*, which specifies the semantics of communication. Currently defined types are:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM

A **SOCK_STREAM** type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A **SOCK_DGRAM** socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). A **SOCK_SEQPACKET** socket may provide a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer may be required to read an entire packet with each read system call. This facility is protocol specific, and presently not implemented for any protocol family. **SOCK_RAW** sockets provide access to internal network interfaces. The types **SOCK_RAW**, which is available only to the super-user, and **SOCK_RDM**, for which no implementation currently exists, are not described here.

The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the "communication domain" in which communication is to take place; see **protocols**(5).

Sockets of type **SOCK_STREAM** are full-duplex byte streams, similar to pipes. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with a **connect**(2) call. Once connected, data may be transferred using **read**(2V) and **write**(2V) calls or some variant of the **send**(2) and **recv**(2) calls. When a session has been completed a **close**(2V), may be performed. Out-of-band data may also be transmitted as described in **send**(2) and received as described in **recv**(2).

The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with −1 returns and with ETIMEDOUT as the specific code in the global variable **errno**. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (for instance 5 minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_SEQPACKET sockets employ the same system calls as SOCK_STREAM sockets. The only difference is that **read**(2V) calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in **send**(2) calls. Datagrams are generally received with **recv**(2), which returns the next datagram with its return address.

An **fcntl**(2V) call can be used to specify a process group to receive a SIGURG signal when the out-of-band data arrives. It may also enable non-blocking I/O and asynchronous notification of I/O events with SIGIO signals.

The operation of sockets is controlled by socket level *options*. These options are defined in the file **socket.h**. **getsockopt**(2) and **setsockopt**( ) are used to get and set options, respectively.

RETURN VALUES

socket( ) returns a non-negative descriptor on success. On failure, it returns −1 and sets **errno** to indicate the error.

ERRORS

EACCES	Permission to create a socket of the specified type and/or protocol is denied.
EMFILE	The per-process descriptor table is full.
ENFILE	The system file table is full.
ENOBUFS	Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.
EPROTONOSUPPORT	The protocol type or the specified protocol is not supported within this domain.
EPROTOTYPE	The protocol is the wrong type for the socket.

SEE ALSO

accept(2), **bind**(2), **close**(2V), **connect**(2), **fcntl**(2V), **getsockname**(2), **getsockopt**(2), **ioctl**(2), **listen**(2), **read**(2V), **recv**(2), **select**(2), **send**(2), **shutdown**(2), **socketpair**(2), **write**(2V), **protocols**(5)

Network Programming

NAME

socketpair – create a pair of connected sockets

SYNOPSIS

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

int socketpair(d, type, protocol, sv)
int d, type, protocol;
int sv[2];
```

DESCRIPTION

The **socketpair( )** system call creates an unnamed pair of connected sockets in the specified address family *d*, of the specified *type* and using the optionally specified *protocol*. The descriptors used in referencing the new sockets are returned in *sv*[0] and *sv*[1]. The two sockets are indistinguishable.

RETURN VALUES

socketpair( ) returns:

0	on success.
–1	on failure and sets **errno** to indicate the error.

ERRORS

EAFNOSUPPORT	The specified address family is not supported on this machine.
EFAULT	The address *sv* does not specify a valid part of the process address space.
EMFILE	Too many descriptors are in use by this process.
EOPNOSUPPORT	The specified protocol does not support creation of socket pairs.
EPROTONOSUPPORT	The specified protocol is not supported on this machine.

SEE ALSO

pipe(2V), **read**(2V), **write**(2V)

BUGS

This call is currently implemented only for the **AF_UNIX** address family.

NAME

stat, lstat, fstat – get file status

SYNOPSIS

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

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

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

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

DESCRIPTION

stat( ) obtains information about the file named by *path*. 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.

lstat( ) is like **stat**( ) except in the case where the named file is a symbolic link, in which case **lstat**( ) returns information about the link, while **stat**( ) returns information about the file the link references.

fstat( ) obtains the same information about an open file referenced by the argument descriptor, such as would be obtained by an **open**(2V) call.

buf is a pointer to a **stat** structure into which information is placed concerning the file. A **stat** structure includes the following members:

```
dev_t     st_dev;     /* device file resides on */
ino_t     st_ino;     /* the file serial number */
mode_t    st_mode;    /* file mode */
nlink_t   st_nlink;   /* number of hard links to the file */
uid_t     st_uid;     /* user ID of owner */
gid_t     st_gid;     /* group ID of owner */
dev_t     st_rdev;    /* the device identifier (special files only)*/
off_t     st_size;    /* total size of file, in bytes */
time_t    st_atime;   /* file last access time */
time_t    st_mtime;   /* file last modify time */
time_t    st_ctime;   /* file last status change time */
long      st_blksize; /* preferred blocksize for file system I/O*/
long      st_blocks;  /* actual number of blocks allocated */
```

st_atime Time when file data was last accessed. This can also be set explicitly by **utimes**(2). **st_atime** is not updated for directories searched during pathname resolution.

st_mtime Time when file data was last modified. This can also be set explicitly by **utimes**(2). It is not set by changes of owner, group, link count, or mode.

st_ctime Time when file status was last changed. It is set both both by writing and changing the file status information, such as changes of owner, group, link count, or mode.

The following macros test whether a file is of the specified type. The value *m* is the value of **st_mode**. Each macro evaluates to a non-zero value if the test is true or to zero if the test is false.

S_ISDIR(*m*) Test for directory file.

S_ISCHR(*m*) Test for character special file.

S_ISBLK(*m*) Test for block special file.

S_ISREG(*m*) Test for regular file.

S_ISLNK(*m*) Test for a symbolic link.

S_ISSOCK(*m*) Test for a socket.

S_ISFIFO(*m*) Test for pipe or FIFO special file.

The status information word **st_mode** is bit-encoded using the following masks and bits:

S_IRWXU Read, write, search (if a directory), or execute (otherwise) permissions mask for the owner of the file.

- **S_IRUSR** Read permission bit for the owner of the file.
- **S_IWUSR** Write permission bit for the owner of the file.
- **S_IXUSR** Search (if a directory) or execute (otherwise) permission bit for the owner of the file.

S_IRWXG Read, write, search (if directory), or execute (otherwise) permissions mask for the file group class.

- **S_IRGRP** Read permission bit for the file group class.
- **S_IWGRP** Write permission bit for the file group class.
- **S_IXGRP** Search (if a directory) or execute (otherwise) permission bit for the file group class.

S_IRWXO Read, write, search (if a directory), or execute (otherwise) permissions mask for the file other class.

- **S_IROTH** Read permission bit for the file other class.
- **S_IWOTH** Write permission bit for the file other class.
- **S_IXOTH** Search (if a directory) or execute (otherwise) permission bit for the file other class.

S_ISUID Set user ID on execution. The process's effective user ID is set to that of the owner of the file when the file is run as a program (see **execve**(2V)). On a regular file, this bit should be cleared on any write.

S_ISGID Set group ID on execution. The process's effective group ID is set to that of the file when the file is run as a program (see **execve**(2V)). On a regular file, this bit should be cleared on any write.

In addition, the following bits and masks are made available for backward compatibility:

```
#define  S_IFMT     0170000   /* type of file */
#define  S_IFIFO    0010000   /* FIFO special */
#define  S_IFCHR    0020000   /* character special */
#define  S_IFDIR    0040000   /* directory */
#define  S_IFBLK    0060000   /* block special */
#define  S_IFREG    0100000   /* regular file */
#define  S_IFLNK    0120000   /* symbolic link */
#define  S_IFSOCK   0140000   /* socket */
#define  S_ISVTX    0001000   /* save swapped text even after use */
#define  S_IREAD    0000400   /* read permission, owner */
#define  S_IWRITE   0000200   /* write permission, owner */
#define  S_IEXEC    0000100   /* execute/search permission, owner */
```

For more information on **st_mode** bits see **chmod**(2V).

RETURN VALUES

stat( ), lstat( ) and **fstat( )** return:

0 on success.

−1 on failure and set **errno** to indicate the error.

ERRORS

stat( ) and **lstat( )** will fail if one or more of the following are true:

EACCES	Search permission is denied for a component of the path prefix of *path*.
EFAULT	*buf* or *path* points to an invalid address.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.n A pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	The file referred to by *path* does not exist.
ENOTDIR	A component of the path prefix of *path* is not a directory.

fstat( ) will fail if one or more of the following are true:

EBADF	*fd* is not a valid open file descriptor.
EFAULT	*buf* points to an invalid address.
EIO	An I/O error occurred while reading from or writing to the file system.

SYSTEM V ERRORS

In addition to the above, the following may also occur:

ENOENT	*path* points to an empty string.

WARNINGS

The **st_atime** and **st_mtime** fields of the **stat( )** are *not* contiguous. Programs that depend on them being contiguous (in calls to **utimes**(2) or **utime**(3V)) will not work.

SEE ALSO

chmod(2V), **chown**(2V), **link**(2V), **open**(2V), **read**(2V), **readlink**(2), **rename**(2V), **truncate**(2), **unlink**(2V), **utimes**(2), **write**(2V)

NAME

statfs, fstatfs – get file system statistics

SYNOPSIS

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

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

int fstatfs(fd, buf)
int fd;
struct statfs *buf;
```

DESCRIPTION

statfs( ) returns information about a mounted file system. *path* is the path name of any file within the mounted filesystem. *buf* is a pointer to a **statfs( )** structure defined as follows:

```
typedef struct {
        long    val[2];
} fsid_t;

struct statfs {
        long    f_type;     /* type of info, zero for now */
        long    f_bsize;    /* fundamental file system block size */
        long    f_blocks;   /* total blocks in file system */
        long    f_bfree;    /* free blocks */
        long    f_bavail;   /* free blocks available to non-super-user */
        long    f_files;    /* total file nodes in file system */
        long    f_ffree;    /* free file nodes in fs */
        fsid_t  f_fsid;     /* file system id */
        long    f_spare[7]; /* spare for later */
};
```

Fields that are undefined for a particular file system are set to –1. **fstatfs( )** returns the same information about an open file referenced by descriptor *fd*.

RETURN VALUES

statfs( ) and **fstatfs( )** return:

0 on success.

–1 on failure and set **errno** to indicate the error.

ERRORS

statfs( ) fails if one or more of the following are true:

EACCES	Search permission is denied for a component of the path prefix of *path*.
EFAULT	*buf* or *path* points to an invalid address.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} (see **sysconf**(2V)) while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	The file referred to by *path* does not exist.
ENOTDIR	A component of the path prefix of *path* is not a directory.

fstatfs( ) fails if one or more of the following are true:

EBADF	*fd* is not a valid open file descriptor.
EFAULT	*buf* points to an invalid address.
EIO	An I/O error occurred while reading from the file system.

BUGS

The NFS revision 2 protocol does not permit the number of free files to be provided to the client; thus, when **statfs( )** or **fstatfs( )** are done on a file on an NFS file system, **f_files** and **f_ffree** are always –1.

NAME

swapon – add a swap device for interleaved paging/swapping

SYNOPSIS

```
int swapon(special)
char *special;
```

DESCRIPTION

swapon() makes the block device *special* available to the system for allocation for paging and swapping. The names of potentially available devices are known to the system and defined at system configuration time. The size of the swap area on *special* is calculated at the time the device is first made available for swapping.

RETURN VALUES

swapon() returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

EACCES	Search permission is denied for a component of the path prefix of *special*.
EBUSY	The device referred to by *special* has already been made available for swapping.
EFAULT	*special* points outside the process's address space.
EIO	An I/O error occurred while reading from or writing to the file system.
	An I/O error occurred while opening the swap device.
ELOOP	Too many symbolic links were encountered in translating *special*.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} (see **sysconf**(2V)) while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENODEV	The device referred to by *special* was not configured into the system as a swap device.
ENOENT	The device referred to by *special* does not exist.
ENOTBLK	The file referred to by *special* is not a block device.
ENOTDIR	A component of the path prefix of *special* is not a directory.
ENXIO	The major device number of the device referred to by *special* is out of range (this indicates no device driver exists for the associated hardware).
EPERM	The caller is not the super-user.

SEE ALSO

fstab(5), **config**(8), **swapon**(8)

BUGS

There is no way to stop swapping on a disk so that the pack may be dismounted.

This call will be upgraded in future versions of the system.

NAME

symlink – make symbolic link to a file

SYNOPSIS

int symlink(name1, name2)
char *name1, *name2;

DESCRIPTION

A symbolic link *name2* is created to *name1* (*name2* is the name of the file created, *name1* is the string used in creating the symbolic link). Either name may be an arbitrary path name; the files need not be on the same file system.

The file that the symbolic link points to is used when an **open**(2V) operation is performed on the link. A **stat**(2V), on a symbolic link returns the linked-to file, while an **lstat**( ) (refer to **stat**(2V)) returns information about the link itself. This can lead to surprising results when a symbolic link is made to a directory. To avoid confusion in programs, the **readlink**(2) call can be used to read the contents of a symbolic link.

RETURN VALUES

symlink( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

The symbolic link is made unless one or more of the following are true:

EACCES	Search permission is denied for a component of the path prefix of *name2*.
EDQUOT	The directory in which the entry for the new symbolic link is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
	The new symbolic link cannot be created because the user's quota of disk blocks on the file system which will contain the link has been exhausted.
	The user's quota of inodes on the file system on which the file is being created has been exhausted.
EEXIST	The file referred to by *name2* already exists.
EFAULT	*name1* or *name2* points outside the process's allocated address space.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *name2*.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} (see **sysconf**(2V)) while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	A component of the path prefix of *name2* does not exist.
ENOSPC	The directory in which the entry for the new symbolic link is being placed cannot be extended because there is no space left on the file system containing the directory.
	The new symbolic link cannot be created because there is no space left on the file system which will contain the link.
	There are no free inodes on the file system on which the file is being created.
ENOTDIR	A component of the path prefix of *name2* is not a directory.
EROFS	The file *name2* would reside on a read-only file system.

SEE ALSO

ln(1V), **link**(2V), **readlink**(2), **unlink**(2V)

NAME

sync – update super-block

SYNOPSIS

sync()

DESCRIPTION

sync() writes out all information in core memory that should be on disk. This includes modified super blocks, modified inodes, and delayed block I/O.

sync() should be used by programs that examine a file system, for example **fsck**(8), **df**(1V), etc. **sync()** is mandatory before a boot.

SEE ALSO

fsync(2), **cron**(8)

BUGS

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

NAME

syscall – indirect system call

SYNOPSIS

#include <sys/syscall.h>

int syscall(number[, arg, ...])
int number;

DESCRIPTION

syscall( ) performs the system call whose assembly language interface has the specified *number*, and arguments *arg* .... Symbolic constants for system calls can be found in the header file **<sys/syscall.h>**.

RETURN VALUES

syscall( ) returns the return value of the system call specified by *number*.

SEE ALSO

intro(2), **pipe**(2V)

WARNINGS

There is no way to use **syscall( )** to call functions such as **pipe**(2V), which return values that do not fit into one hardware register.

Since many system calls are implemented as library wrappers around traps to the kernel, these calls may not behave as documented when called from **syscall( )**, which bypasses these wrappers. For these reasons, using **syscall( )** is not recommended.

NAME

sysconf – query system related limits, values, options

SYNOPSIS

```
#include <unistd.h>

long sysconf(name)
int name;
```

DESCRIPTION

The **sysconf()** function provides a method for the application to determine the current value of a configurable system limit or option (variable). The value does not change during the lifetime of the calling process.

The convention used throughout sections 2 and 3 is that {LIMIT} means that LIMIT is something that can change from system to system and applications that want accurate values need to call **sysconf()**. These values are things that have been historically available in header files such as **<sys/param.h>**.

The following lists the conceptual name and meaning of each variable.

Name	*Meaning*
{ARG_MAX}	Max combined size of **argv[ ]** & **envp[ ]**.
{CHILD_MAX}	Max processes allowed to any UID.
{CLK_TCK}	Ticks per second (**clock_t**).
{NGROUPS_MAX}	Max simultaneous groups one may belong to.
{OPEN_MAX}	Max open files per process.
{_POSIX_JOB_CONTROL}	Job control supported (boolean).
{_POSIX_SAVED_IDS}	Saved ids (**seteuid()**) supported (boolean).
{_POSIX_VERSION}	Version of the POSIX.1 standard supported.

The following table lists the conceptual name of each variable and the flag passed to **sysconf()** to retrieve the value of each variable.

Name	*Sysconf flag*
{ARG_MAX}	**_SC_ARG_MAX**
{CHILD_MAX}	**_SC_CHILD_MAX**
{CLK_TCK}	**_SC_CLK_TCK**
{NGROUPS_MAX}	**_SC_NGROUPS_MAX**
{OPEN_MAX}	**_SC_OPEN_MAX**
{_POSIX_JOB_CONTROL}	**_SC_JOB_CONTROL**
{_POSIX_SAVED_IDS}	**_SC_SAVED_IDS**
{_POSIX_VERSION}	**_SC_VERSION**

RETURN VALUES

sysconf() returns the current variable value on success. On failure, it returns –1 and sets **errno** to indicate the error.

ERRORS

EINVAL The value of *name* is invalid.

NAME

truncate, ftruncate – set a file to a specified length

SYNOPSIS

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

int truncate(path, length)
char *path;
off_t length;

int ftruncate(fd, length)
int fd;
off_t length;
```

DESCRIPTION

truncate( ) causes the file referred to by *path* (or for **ftruncate( )** the object referred to by *fd*) to have a size equal to *length* bytes. If the file was previously longer than *length*, the extra bytes are removed from the file. If it was shorter, bytes between the old and new lengths are read as zeroes. With **ftruncate( )**, the file must be open for writing.

RETURN VALUES

truncate( ) returns:

0	on success.
−1	on failure and sets **errno** to indicate the error.

ERRORS

truncate( ) may set **errno** to:

EACCES	Search permission is denied for a component of the path prefix of *path*.
	Write permission is denied for the file referred to by *path*.
EFAULT	*path* points outside the process's allocated address space.
EIO	An I/O error occurred while reading from or writing to the file system.
EISDIR	The file referred to by *path* is a directory.
ELOOP	Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} (see **sysconf**(2V)) while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	The file referred to by *path* does not exist.
ENOTDIR	A component of the path prefix of *path* is not a directory.
EROFS	The file referred to by *path* resides on a read-only file system.

ftruncate( ) may set **errno** to:

EINVAL	*fd* is not a valid descriptor of a file open for writing.
	fd refers to a socket, not to a file.
EIO	An I/O error occurred while reading from or writing to the file system.

SEE ALSO

open(2V)

BUGS

These calls should be generalized to allow ranges of bytes in a file to be discarded.

NAME

umask – set file creation mode mask

SYNOPSIS

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

int umask(mask)
int mask;
```

SYSTEM V SYNOPSIS

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

mode_t umask(mask)
mode_t mask;
```

DESCRIPTION

umask() sets the process's file creation mask to *mask* and returns the previous value of the mask. The low-order 9 bits of *mask* are used whenever a file is created, clearing corresponding bits in the file access permissions. (see **stat**(2V)). This clearing restricts the default access to a file.

The mask is inherited by child processes.

RETURN VALUES

umask() returns the previous value of the file creation mask.

SEE ALSO

chmod(2V), **mknod**(2V), **open**(2V)

NAME

uname – get information about current system

SYNOPSIS

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

int uname (name)
struct utsname *name;
```

DESCRIPTION

uname( ) stores information identifying the current operating system in the structure pointed to by *name*.

uname( ) uses the structure defined in **<sys/utsname.h>**, the members of which are:

```
struct utsname {
        char    sysname[9];
        char    nodename[9];
        char    nodeext[65-9];
        char    release[9];
        char    version[9];
        char    machine[9];
}
```

uname( ) places a null-terminated character string naming the current operating system in the character array *sysname*; this string is "SunOS" on Sun systems. *nodename* is set to the name that the system is known by on a communications network; this is the same value as is returned by **gethostname**(2). *release* and *version* are set to values that further identify the operating system. *machine* is set to a standard name that identifies the hardware on which the SunOS system is running. This is the same as the value displayed by **arch**(1).

RETURN VALUES

uname( ) returns:

0 on success.

–1 on failure.

SEE ALSO

arch(1), **uname**(1), **gethostname**(2)

NOTES

nodeext is provided for backwards compatability with previous SunOS Releases and provides space for node names longer than eight bytes. Applications should not use *nodeext*. To be maximally portable, applications that want to copy the node name to another string should use **strlen(nodename)** rather than the constant 9 or **sizeof(nodename)** as the size of the target string.

System administrators should note that systems with node names longer than eight bytes do not conform to *IEEE Std 1003.1-1988*, *System V Interface Definition* (Issue 2), or *X/Open Portability Guide* (Issue 2) requirements.

NAME

unlink – remove directory entry

SYNOPSIS

int unlink(path)
char *path;

DESCRIPTION

unlink() removes the directory entry named by the pathname pointed to by *path* and decrements the link count of the file referred to by that entry. If this entry was the last link to the file, and no process has the file open, then all resources associated with the file are reclaimed. If, however, the file was open in any process, the actual resource reclamation is delayed until it is closed, even though the directory entry has disappeared.

If *path* refers to a directory, the effective user-ID of the calling process must be super-user.

Upon successful completion, **unlink()** marks for update the **st_ctime** and **st_mtime** fields of the parent directory. Also, if the file's link count is not zero, the **st_ctime** field of the file is marked for update.

RETURN VALUES

unlink() returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

EACCES	Search permission is denied for a component of the path prefix of *path*.
	Write permission is denied for the directory containing the link to be removed.
EBUSY	The entry to be unlinked is the mount point for a mounted file system.
EFAULT	*path* points outside the process's allocated address space.
EINVAL	The file referred to by *path* is the current directory, '.'.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	The file referred to by *path* does not exist.
ENOTDIR	A component of the path prefix of *path* is not a directory.
EPERM	The file referred to by *path* is a directory and the effective user ID of the process is not the super-user.
EROFS	The file referred to by *path* resides on a read-only file system.

SYSTEM V ERRORS

In addition to the above, the following may also occur:

ENOENT	*path* points to an empty string.

SEE ALSO

close(2V), **link**(2V), **rmdir**(2V)

NOTES

Applications should use **rmdir**(2V) to remove directories. Although **root** may use **unlink()** on directories, all users may use **rmdir()**.

NAME

unmount, umount – remove a file system

SYNOPSIS

int unmount(name)
char *name;

SYSTEM V SYNOPSIS

int umount(special)
char *special;

DESCRIPTION

unmount() announces to the system that the directory *name* is no longer to refer to the root of a mounted file system. The directory *name* reverts to its ordinary interpretation.

Only the super-user may call **unmount()**.

SYSTEM V DESCRIPTION

umount() reqests that a previously mounted file system contained on the block special device referred to by *special* be unmounted. *special* points to a path name. After the file system is unmounted, the directory on which it was mounted reverts to its ordinary interpretation.

Only the super-user may call **umount()**.

Note: Unlike the path name argument to **unmount()** which refers to the directory on which the file system is mounted, *special* refers to the block special device containing the mounted file system itself.

RETURN VALUES

unmount() returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

SYSTEM V RETURN VALUES

umount() returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

EACCES	Search permission is denied for a component of the path prefix.
EBUSY	A process is holding a reference to a file located on the file system.
EFAULT	*name* points outside the process's allocated address space.
EINVAL	*name* is not the root of a mounted file system.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating the path name.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} (see **sysconf**(2V)) while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	*name* does not exist.
ENOTDIR	A component of the path prefix of *name* is not a directory.
EPERM	The caller is not the super-user.

SYSTEM V ERRORS

EINVAL	The device referred to by *special* is not mounted.
ENOENT	The named file does not exist.

ENOTBLK	*special* does not refer to a block special file.
ENOTDIR	A component of the path prefix of *special* is not a directory.
ENXIO	The device referred to by *special* does not exist.

SEE ALSO

mount(2V), **mount**(8)

BUGS

The error codes are in a state of disarray; too many errors appear to the caller as one value.

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. This is normally the value returned in the **st_dev** field of a **stat** structure when a **stat( )**, **fstat( )**, or **lstat( )** call is made on a file on that file system. *buf* is a pointer to a **ustat** structure that includes the following elements:

```
daddr_t  f_tfree;        /* Total blocks available to non-super-user */
ino_t    f_tinode;       /* Number of free files */
char     f_fname[6];     /* Filsys name */
char     f_fpack[6];     /* Filsys pack name */
```

The **f_fname** and **f_fpack** fields are always set to a null string. Other fields that are undefined for a particular file system are set to –1.

RETURN VALUES

ustat( ) returns:

0 on success.

–1 on failure and sets **errno** to indicate the error.

ERRORS

EFAULT *buf* points to an invalid address.

EINVAL *dev* is not the device number of a device containing a mounted file system.

EIO An I/O error occurred while reading from or writing to the file system.

SEE ALSO

stat(2V), **statfs**(2)

BUGS

The NFS revision 2 protocol does not permit the number of free files to be provided to the client; thus, when **ustat( )** is done on an NFS file system, **f_tinode** is always –1.

NAME

utimes – set file times

SYNOPSIS

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

int utimes(file, tvp)
char *file;
struct timeval *tvp;
```

DESCRIPTION

utimes( ) sets the access and modification times of the file named by *file*.

If *tvp* is NULL, the access and modification times are set to the current time. A process must be the owner of the file or have write permission for the file to use **utimes( )** in this manner.

If *tvp* is not NULL, it is assumed to point to an array of two **timeval** structures. The access time is set to the value of the first member, and the modification time is set to the value of the second member. Only the owner of the file or the super-user may use **utimes( )** in this manner.

In either case, the *inode-changed* time of the file is set to the current time.

RETURN VALUES

utimes( ) returns:

0 on success.

−1 on failure and sets **errno** to indicate the error.

ERRORS

EACCES	Search permission is denied for a component of the path prefix of *file*.
EACCES	The effective user ID of the process is not super-user and not the owner of the file, write permission is denied for the file, and *tvp* is NULL.
EFAULT	*file* or *tvp* points outside the process's allocated address space.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *file*.
ENOENT	The file referred to by *file* does not exist.
ENOTDIR	A component of the path prefix of *file* is not a directory.
EPERM	The effective user ID of the process is not super-user and not the owner of the file, and *tvp* is not NULL.
EROFS	The file system containing the file is mounted read-only.

SEE ALSO

stat(2V)

NAME

vadvise – give advice to paging system

SYNOPSIS

#include <sys/vadvise.h>

vadvise(param)
int param;

DESCRIPTION

vadvise( ) is used to inform the system that process paging behavior merits special consideration. Parameters to **vadvise( )** are defined in the file **<sys/vadvise.h>**. Currently, two calls to **vadvise( )** are implemented.

vadvise(VA_ANOM);

advises that the paging behavior is not likely to be well handled by the system's default algorithm, since reference information that is collected over macroscopic intervals (for instance, 10-20 seconds) will not serve to indicate future page references. The system in this case will choose to replace pages with little emphasis placed on recent usage, and more emphasis on referenceless circular behavior. It is *essential* that processes which have very random paging behavior (such as LISP during garbage collection of very large address spaces) call **vadvise**, as otherwise the system has great difficulty dealing with their page-consumptive demands.

vadvise(VA_NORM);

restores default paging replacement behavior after a call to

vadvise(VA_ANOM);

BUGS

The current implementation of **vadvise( )** will go away soon, being replaced by a per-page **vadvise( )** facility.

NAME

vfork – spawn new process in a virtual memory efficient way

SYNOPSIS

#include <vfork.h>

int vfork()

DESCRIPTION

vfork() can be used to create new processes without fully copying the address space of the old process, which is horrendously inefficient in a paged environment. It is useful when the purpose of **fork**(2V), would have been to create a new system context for an **execve**(2V). **vfork()** differs from **fork()** in that the child borrows the parent's memory and thread of control until a call to **execve**(2V), or an exit (either by a call to **exit**(2V) or abnormally.) The parent process is suspended while the child is using its resources.

vfork() returns 0 in the child's context and (later) the process ID (PID) of the child in the parent's context.

vfork() can normally be used just like **fork**. It does not work, however, to return while running in the child's context from the procedure which called **vfork()** since the eventual return from **vfork()** would then return to a no longer existent stack frame. Be careful, also, to call **_exit()** rather than **exit()** if you cannot *execve*, since **exit()** will flush and close standard I/O channels, and thereby mess up the parent processes standard I/O data structures. (Even with **fork()** it is wrong to call **exit()** since buffered data would then be flushed twice.)

On Sun-4 machines, the parent inherits the values of local and incoming argument registers from the child. Since this violates the usual data flow properties of procedure calls, the file **<vfork.h>** must be included in programs that are compiled using global optimization.

RETURN VALUES

On success, **vfork()** returns 0 to the child process and returns the process ID of the child process to the parent process. On failure, **vfork()** returns –1 to the parent process, sets **errno** to indicate the error, and no child process is created.

SEE ALSO

execve(2V), **exit**(2V), **fork**(2V), **ioctl**(2), **sigvec**(2), **wait**(2V)

BUGS

This system call will be eliminated in a future release. System implementation changes are making the efficiency gain of **vfork()** over **fork**(2V) smaller. The memory sharing semantics of **vfork()** can be obtained through other mechanisms.

To avoid a possible deadlock situation, processes that are children in the middle of a **vfork()** are never sent **SIGTTOU** or **SIGTTIN** signals; rather, output or *ioctl*s are allowed and input attempts result in an EOF indication.

NAME

vhangup – virtually ''hangup'' the current control terminal

SYNOPSIS

vhangup()

DESCRIPTION

vhangup() is used by the initialization process **init**(8) (among others) to arrange that users are given "clean" terminals at login, by revoking access of the previous users' processes to the terminal. To affect this, **vhangup()** searches the system tables for references to the control terminal of the invoking process, revoking access permissions on each instance of the terminal that it finds. Further attempts to access the terminal by the affected processes will yield I/O errors (EBADF). Finally, a SIGHUP (hangup signal) is sent to the process group of the control terminal.

SEE ALSO

init(8)

BUGS

Access to the control terminal using **/dev/tty** is still possible.

This call should be replaced by an automatic mechanism that takes place on process exit.

NAME

wait, wait3, wait4, waitpid, WIFSTOPPED, WIFSIGNALED, WIFEXITED, WEXITSTATUS, WTERMSIG, WSTOPSIG – wait for process to terminate or stop, examine returned status

SYNOPSIS

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

int wait(statusp)
int *statusp;

int waitpid(pid, statusp, options)
int pid;
int *statusp;
int options;

#include <sys/time.h>
#include <sys/resource.h>

int wait3(statusp, options, rusage)
int *statusp;
int options;
struct rusage *rusage;

int wait4(pid, statusp, options, rusage)
int pid;
int *statusp;
int options;
struct rusage *rusage;

WIFSTOPPED(status)
int status;

WIFSIGNALED(status)
int status;

WIFEXITED(status)
int status

WEXITSTATUS(status)
int status

WTERMSIG(status)
int status

WSTOPSIG(status)
int status
```

SYSTEM V SYNOPSIS

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

pid_t wait(statusp)
int *statusp;

pid_t waitpid(pid, statusp, options)
pid_t pid;
int *statusp;
int options;
```

DESCRIPTION

wait() delays its caller until a signal is received or one of its child processes terminates or stops due to tracing. If any child has died or stopped due to tracing and this has not been reported using **wait()**, return is immediate, returning the process ID and exit status of one of those children. If that child had died, it is discarded. If there are no children, return is immediate with the value −1 returned. If there are only running or stopped but reported children, the calling process is blocked.

If *statusp* is not a NULL pointer, then on return from a successful **wait()** call the status of the child process whose process ID is the return value of **wait()** is stored in the location pointed to by *statusp*. It indicates the cause of termination and other information about the terminated process in the following manner:

- If the first byte (the low-order 8 bits) are equal to 0177, the child process has stopped. The next byte contains the number of the signal that caused the process to stop. See **ptrace**(2) and **sigvec**(2).
- If the first byte (the low-order 8 bits) are non-zero and are not equal to 0177, the child process terminated due to a signal. The low-order 7 bits contain the number of the signal that terminated the process. In addition, if the low-order seventh bit (that is, bit 0200) is set, a "core image" of the process was produced (see **sigvec**(2)).
- Otherwise, the child process terminated due to a call to **exit**(2V). The next byte contains the low-order 8 bits of the argument that the child process passed to **exit()**.

waitpid() behaves identically to **wait()** if *pid* has a value of −1 and *options* has a value of zero. Otherwise, the behavior of **waitpid()** is modified by the values of *pid* and *options* as follows:

pid specifies a set of child processes for which status is requested. **waitpid()** only returns the status of a child process from this set.

- If *pid* is equal to −1, status is requested for any child process. In this repect, **waitpid()** is then equivalent to **wait()**.
- If *pid* is greater than zero, it specifies the process ID of a single child process for which status is requested.
- If *pid* is equal to zero, status is requested for any child process whose process group ID is equal to that of the calling process.
- If *pid* is less than −1, status is requested for any child process whose process group ID is equal to the absolute value of *pid*.

options is constructed from the bitwise inclusive OR of zero or more of the following flags, defined in the header **<sys/wait.h>**:

WNOHANG
: **waitpid()** does not suspend execution of the calling process if status is not immediately available for one of the child processes specified by *pid*.

WUNTRACED
: The status of any child processes specified by *pid* that are stopped, and whose status has not yet been reported since they stopped, are also reported to the requesting process.

wait3() is an alternate interface that allows both non-blocking status collection and the collection of the status of children stopped by any means. The *status* parameter is defined as above. The *options* parameter is used to indicate the call should not block if there are no processes that have status to report (**WNOHANG**), and/or that children of the current process that are stopped due to a **SIGTTIN**, **SIGTTOU**, **SIGTSTP**, or **SIGSTOP** signal are eligible to have their status reported as well (**WUNTRACED**). A terminated child is discarded after it reports status, and a stopped process will not report its status more than once. If *rusage* is not a NULL pointer, a summary of the resources used by the terminated process and all its children is returned. (This information is currently not available for stopped processes.)

When the WNOHANG option is specified and no processes have status to report, **wait3()** returns 0. The WNOHANG and WUNTRACED options may be combined by ORing the two values.

wait4() is another alternate interface. With a *pid* argument of 0, it is equivalent to **wait3()**. If *pid* has a nonzero value, then **wait4()** returns status only for the indicated process ID, but not for any other child processes.

WIFSTOPPED, WIFSIGNALED, WIFEXITED, WEXITSTATUS, WTERMSIG, and WSTOPSIG are macros that take an argument *status*, of type **'int'**, as returned by **wait()**, **wait3()**, or **wait4()**. WIFSTOPPED evaluates to true (1) when the process for which the **wait()** call was made is stopped, or to false (0) otherwise. If WIFSTOPPED(*status*) is non-zero, WSTOPSIG evaluates to the number of the signal that caused the child process to stop. WIFSIGNALED evaluates to true when the process was terminated with a signal. If WIFSIGNALED(*status*) is non-zero, WTERMSIG evaluates to the number of the signal that caused the termination of the child process. WIFEXITED evaluates to true when the process exited by using an **exit**(2V) call. If WIFEXITED(*status*) is non-zero, WEXITSTATUS evaluates to the low-order byte of the argument that the child process passed to **_exit()** (see **exit**(2V)) or **exit**(3), or the value the child process returned from **main()** (see **execve**(2V)).

If the information stored at the location pointed to by *statusp* was stored there by a call to **waitpid()** that specified the WUNTRACED flag, exactly one of the macros WIFEXITED(**statusp*), WIFSIGNALED(**statusp*), and WIFSTOPPED(**statusp*) will evaluate to a non-zero value. If the information stored at the location pointed to by *statusp* was stored there by a call to **waitpid()** that did *not* specify the WUNTRACED flag or by a call to **wait()**, exactly one of the macros WIFEXITED(**statusp*) and WIFSIGNALED(**statusp*) will evaluate to a non-zero value.

If a parent process terminates witout waiting for all of its child processes to terminate, the remaining child processes are assigned the parent process ID of 1, corresponding to **init**(8).

RETURN VALUES

If **wait()** or **waitpid()** 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.

If **wait()** or **waitpid()** return due to the delivery of a signal to the calling process, a value of −1 is returned and **errno** is set to EINTR. If **waitpid()** function was invoked with WNOHANG set in *options*, it has at least one child process specified by *pid* for which status is not available, and status is not available for any process specified by *pid*, a value of zero is returned. Otherwise, a value of −1 is returned, and **errno** is set to indicate the error.

wait3() and **wait4()** return 0 if WNOHANG is specified and there are no stopped or exited children, and return the process ID of the child process if they return due to a stopped or terminated child process. Otherwise, they return a value of −1 and set **errno** to indicate the error.

ERRORS

wait(), **wait3()**, or **wait4()** will fail and return immediately if one or more of the following are true:

ECHILD	The calling process has no existing unwaited-for child processes.
EFAULT	*statusp* or *rusage* points to an illegal address.
EINTR	The function was interrupted by a signal. The value of the location pointed to by *statusp* is undefined.

waitpid() may set **errno** to:

ECHILD	The process or process group specified by *pid* does not exist or is not a child of the calling process.
EINTR	The function was interrupted by a signal. The value of the location pointed to by *statusp* is undefined.
EINVAL	The value of *options* is not valid.

wait(), **wait3()**, and **wait4()** will terminate prematurely, return −1, and set **errno** to: EINTR upon the arrival of a signal whose **SV_INTERRUPT** bit in its flags field is set (see **sigvec**(2) and **siginterrupt**(3V)). **signal**(3V), in the System V compatibility library, sets this bit for any signal it catches.

SEE ALSO

exit(2V), **fork**(2V), **getrusage**(2), **ptrace**(2), **sigvec**(2), **pause**(3V), **siginterrupt**(3V), **signal**(3V), **times**(3V)

NOTES

If a parent process terminates without waiting on its children, the initialization process (process ID = 1) inherits the children.

wait(), **wait3()**, and **wait4()** are automatically restarted when a process receives a signal while awaiting termination of a child process, unless the **SV_INTERRUPT** bit is set in the flags for that signal.

Previous SunOS releases used **union wait *statusp** and **union wait status** in place of **int *statusp** and **intstatus.** The union contained a member **w_status** that could be treated in the same way as *status*.

Other members of the **wait** union could be used to extract this information more conveniently:

- If the **w_stopval** member had the value **WSTOPPED**, the child process had stopped; the value of the **w_stopsig** member was the signal that stopped the process.
- If the **w_termsig** member was non-zero, the child process terminated due to a signal; the value of the **w_termsig** member was the number of the signal that terminated the process. If the **w_coredump** member was non-zero, a core dump was produced.
- Otherwise, the child process terminated due to a call to **exit()**. The value of the **w_retcode** member was the low-order 8 bits of the argument that the child process passed to **exit()**.

union wait is obsolete in light of the new specifications provided by *IEEE Std 1003.1-1988* and endorsed by *SVID89* and *XPG3*. SunOS Release 4.1 supports **union wait** for backward compatibility, but it will disappear in a future release.

NAME

write, writev – write output

SYNOPSIS

```
int write(fd, buf, nbyte)
int fd;
char *buf;
int nbyte;

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

int writev(fd, iov, iovcnt)
int fd;
struct iovec *iov;
int iovcnt;
```

SYSTEM V SYNOPSIS

```
int write(fd, buf, nbyte)
int fd;
char *buf;
unsigned nbyte;
```

DESCRIPTION

write( ) attempts to write *nbyte* bytes of data to the object referenced by the descriptor *fd* from the buffer pointed to by *buf*. **writev( )** performs the same action, but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0],*iov*[1], ..., *iov*[*iovcnt* – 1]. If *nbyte* is zero, **write( )** takes no action and returns 0. **writev( )**, however, returns –1 and sets the global variable **errno** (see ERRORS below).

For **writev( )**, the **iovec** structure is defined as

```
struct iovec {
        caddr_t iov_base;
        int     iov_len;
};
```

Each **iovec** entry specifies the base address and length of an area in memory from which data should be written. **writev( )** always writes a complete area before proceeding to the next.

On objects capable of seeking, the **write( )** starts at a position given by the seek pointer associated with *fd*, (see **lseek**(2V)). Upon return from **write( )**, the seek pointer is incremented by the number of bytes actually written.

Objects that are not capable of seeking always write from the current position. The value of the seek pointer associated with such an object is undefined.

If the **O_APPEND** flag of the file status flags is set, the seek pointer is set to the end of the file prior to each write.

If the process calling **write( )** or **writev( )** receives a signal before any data are written, the system call is restarted, unless the process explicitly set the signal to interrupt the call using **sigvec( )** or **sigaction( )** (see the discussions of **SV_INTERRUPT** on **sigvec**(2) and **SA_INTERRUPT** on **sigaction**(3V)). If **write( )** or **writev( )** is interrupted by a signal after successfully writing some data, it returns the number of bytes written.

For regular files, if the **O_SYNC** flag of the file status flags is set, **write( )** does 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( )** does not return until the data has been physically updated.

If the real user is not the super-user, then **write( )** clears the set-user-id bit on a file. This prevents penetration of system security by a user who "captures" a writable set-user-id file owned by the super-user.

For STREAMS (see **intro**(2)) files, the operation of **write( )** and **writev( )** are determined by the values of the minimum and maximum packet sizes accepted by the *stream*. These values are contained in the topmost *stream* module. Unless the user pushes (see **I_PUSH** in **streamio**(4)) the topmost module, these values can not be set or tested from user level. If the total number of bytes to be written falls within the packet size range, that many bytes are written. If the total number of bytes to be written does not fall within the range and the minimum packet size value is zero, **write( )** and **writev( )** break the data to be written into maximum packet size segments prior to sending the data downstream (the last segment may contain less than the maximum packet size). If the total number of bytes to be written does not fall within the range and the minimum value is non-zero, **write( )** and **writev( )** fail and set **errno** to ERANGE. Writing a zero-length buffer (the total number of bytes to be written is zero) sends zero bytes with zero returned.

When a descriptor or the object it refers to is marked for non-blocking I/O, and the descriptor refers to an object subject to flow control, such as a socket, a pipe (or FIFO), or a *stream*, **write( )** and **writev( )** may write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible. If such an object's buffers are full, so that it cannot accept any data, then:

- If the object to which the descriptor refers is marked for non-blocking I/O using the **FIONBIO** request to **ioctl**(2), or by using **fcntl**(2V) to set the **FNDELAY** or **O_NDELAY** flag (defined in **<sys/fcntl.h>**), **write( )** returns −1 and sets **errno** to EWOULDBLOCK.

Upon successful completion, **write( )** marks for update the **st_ctime** and **st_mtime** fields of the file.

SYSTEM V DESCRIPTION

write( ) and **writev( )** behave as described above, except:

When a descriptor or the object it refers to is marked for non-blocking I/O, and the descriptor refers to an object subject to flow control, such as a socket, a pipe (or FIFO), or a *stream*, **write( )** and **writev( )** may write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible. If such an object's buffers are full, so that it cannot accept any data, then:

- If the descriptor is marked for non-blocking I/O by using **fcntl( )** to set the **FNBIO** or **O_NDELAY** flag (defined in **<sys/fcntl.h>**), and does not refer to a *stream*, the **write( )** returns 0. If the descriptor is marked for non-blocking I/O, and refers to a *stream*, **write( )** returns −1 and sets **errno** to EAGAIN.
- If the descriptor is marked for non-blocking I/O using **fcntl( )** to set the **FNONBLOCK** or **O_NONBLOCK** flag (defined in **<sys/fcntl.h>**), **write( )** requests for {PIPE_BUF} (see **pathconf**(2V)) or fewer bytes either succeed completely and return *nbyte*, or return −1 and set **errno** to EAGAIN. A **write( )** request for greater than {PIPE_BUF} bytes either transfers what it can and returns the number of bytes written, or transfers no data and returns −1 and sets **errno** to EAGAIN. If a **write( )** request is greater than {PIPE_BUF} bytes and all data previously written to the pipe has been read, **write( )** transfers at least {PIPE_BUF} bytes.

RETURN VALUES

write( ) and **writev( )** return the number of bytes actually written on success. On failure, they return −1 and set **errno** to indicate the error.

ERRORS

write( ) and **writev( )** fail and the seek pointer remains unchanged if one or more of the following are true:

EBADF	*fd* is not a valid descriptor open for writing.
EDQUOT	The user's quota of disk blocks on the file system containing the file has been exhausted.
EFAULT	Part of *iov* or data to be written to the file 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.

EINTR — The process performing a write received a signal before any data were written, and the signal was set to interrupt the system call.

EINVAL — The *stream* is linked below a multiplexor.

The seek pointer associated with *fd* was negative.

EIO — An I/O error occurred while reading from or writing to the file system.

The process is in a background process group and is attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the process group of the process is orphaned.

ENOSPC — There is no free space remaining on the file system containing the file.

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

EPIPE — An attempt is made to write to a pipe that is not open for reading by any process (or to a socket of type **SOCK_STREAM** that is connected to a peer socket.) Note: an attempted write of this kind also causes you to receive a **SIGPIPE** signal from the kernel. If you've not made a special provision to catch or ignore this signal, then your process dies.

ERANGE — *fd* refers to a *stream*, the total number of bytes to be written is outside the minimum and maximum write range, and the minimum value is non-zero.

EWOULDBLOCK — The file was marked for non-blocking I/O, and no data could be written immediately.

In addition to the above, **writev()** may set **errno** to:

EINVAL — *iovcnt* was less than or equal to 0, or greater than 16.

One of the **iov_len** values in the *iov* array was negative.

The sum of the **iov_len** values in the *iov* array overflowed a 32-bit integer.

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.

SYSTEM V ERRORS

write() fails and sets **errno** as described above, except:

EAGAIN — The descriptor referred to a *stream*, was marked for non-blocking I/O, and no data could be written immediately.

The **O_NONBLOCK** flag is set for the file descriptor and **write()** would block.

SEE ALSO

dup(2V), **fcntl**(2V), **intro**(2), **ioctl**(2), **lseek**(2V), **open**(2V), **pipe**(2V), **select**(2), **sigvec**(2), **signal**(3V)