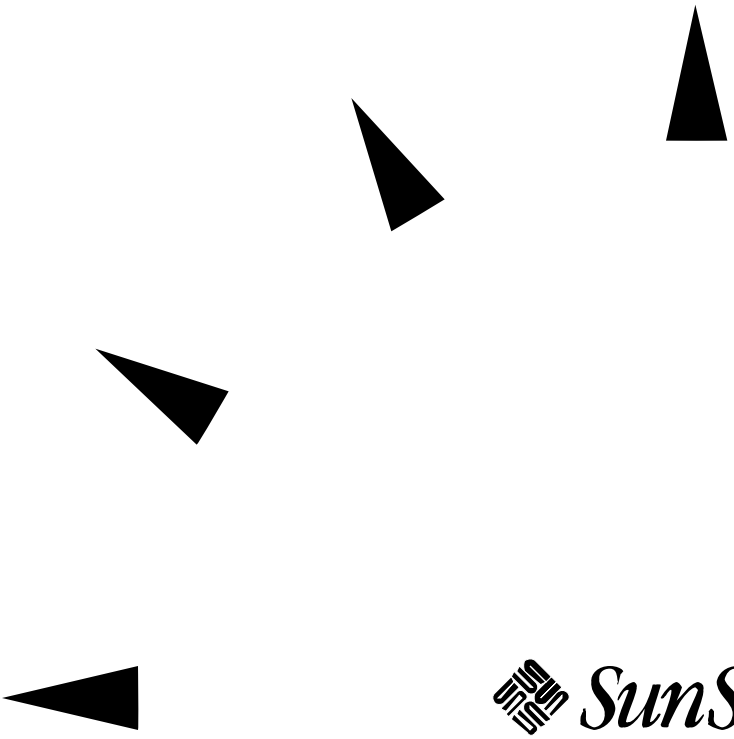
SunOS Reference Manual



Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

SunSoft
A Sun Microsystems, Inc. Business

Preface

OVERVIEW

A man page is provided for both the naive user, and sophisticated user who is familiar with the SunOS operating system and is in need of on-line information. A man page is intended to answer concisely the question “What does it do?” The man pages in general comprise a reference manual. They are not intended to be a tutorial.

The following contains a brief description of each section in the man pages and the information it references:

- Section 1 describes, in alphabetical order, commands available with the operating system.

- Section 1M describes, in alphabetical order, commands that are used chiefly for system maintenance and administration purposes.

- Section 2 describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value.

- Section 3 describes functions found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2 of this volume.

- Section 4 outlines the formats of various files. The C structure declarations for the file formats are given where applicable.

- Section 5 contains miscellaneous documentation such as character set tables, etc.

- Section 6 contains available games and demos.

- Section 7 describes various special files that refer to specific hardware peripherals, and device drivers. STREAMS software drivers, modules and the STREAMS-generic set of system calls are also described.

- Section 9 provides reference information needed to write device drivers in the kernel operating systems environment. It describes two device driver interface specifications: the Device Driver Interface (DDI) and the Driver–Kernel Interface (DKI).

- Section 9E describes the DDI/DKI, DDI-only, and DKI-only entry-point routines a developer may include in a device driver.

- Section 9F describes the kernel functions available for use by device drivers.

- Section 9S describes the data structures used by drivers to share information between the driver and the kernel.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report, there is no BUGS section. See the intro pages for more information and detail about each section, and **man**(1) for more information about man pages in general.

NAME

This section gives the names of the commands or functions documented, followed by a brief description of what they do.

SYNOPSIS

This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full pathname is shown. Literal characters (commands and options) are in **bold** font and variables (arguments, parameters and substitution characters) are in *italic* font. Options and

arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

[ ] The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument *must* be specified.

... Ellipses. Several values may be provided for the previous argument, or the previous argument can be specified multiple times, for example, '*filename* ...'.

| Separator. Only one of the arguments separated by this character can be specified at time.

{ } Braces. The options and/or arguments enclosed within braces are interdependent, such that everything enclosed must be treated as a unit.

PROTOCOL

This section occurs only in subsection 3R to indicate the protocol description file. The protocol specification pathname is always listed in **bold** font.

AVAILABILITY

This section briefly states any limitations on the availabilty of the command. These limitations could be hardware or software specific.

A specification of a class of hardware platform, such as **x86** or **SPARC**, denotes that the command or interface is applicable for the hardware platform specified.

In Section 1 and Section 1M, **AVAILABILITY** indicates which package contains the command being described on the manual page. In order to use the command, the specified package must have been installed with the operating system. If the package was not installed, see **pkgadd**(1) for information on how to upgrade.

MT-LEVEL

This section lists the **MT-LEVEL** of the library functions described in the Section 3 manual pages. The **MT-LEVEL** defines the libraries' ability to support threads. See **Intro**(3) for more information.

DESCRIPTION

This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES. Interactive commands, subcommands, requests, macros, functions and such, are described under USAGE.

IOCTL

This section appears on pages in Section 7 only. Only the device class which supplies appropriate parameters to the **ioctl**(2) system call is called **ioctl** and generates its own heading. **ioctl** calls for a specific device are listed alphabetically (on the man page for that specific device). **ioctl** calls are used for a particular class of devices all of which have an **io** ending, such as **mtio**(7).

OPTIONS

This lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.

OPERANDS

This section lists the command operands and describes how they affect the actions of the command.

OUTPUT

This section describes the output - standard output, standard error, or output files - generated by the command.

RETURN VALUES

If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared as **void** do not return values, so they are not discussed in RETURN VALUES.

ERRORS

On failure, most functions place an error code in the global variable **errno** indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code.

USAGE

This section is provided as a *guidance* on use. This section lists special rules, features and commands that require in-depth explanations. The subsections listed below are used to explain built-in functionality:

Commands
Modifiers
Variables
Expressions
Input Grammar

EXAMPLES

This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command line entry and machine response is shown. Whenever an example is given, the prompt is shown as

example%

or if the user must be super-user,

example#

Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS and USAGE sections.

ENVIRONMENT

This section lists any environment variables that the command or function affects, followed by a brief description of the effect.

EXIT STATUS

This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion and values greater than zero for various error conditions.

FILES

This section lists all filenames referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.

SEE ALSO

This section lists references to other man pages, in-house documentation and outside publications.

DIAGNOSTICS

This section lists diagnostic messages with a brief explanation of the condition causing the error. Messages appear in **bold** font with the exception of variables, which are in *italic* font.

WARNINGS

This section lists warnings about special conditions which could seriously affect your working conditions — this is not a list of diagnostics.

NOTES

This section lists additional information that does not belong anywhere else on the page. It takes the form of an *aside* to the user, covering points of special interest. Critical information is never covered here.

BUGS

This section describes known bugs and wherever possible suggests workarounds.

NAME

Intro, intro – introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value. This is almost always –1 or the null pointer; the individual descriptions specify the details. An error number is also made available in the external variable **errno**. **errno** is not cleared on successful calls, so it should be tested only after an error has been indicated.

In the case of multithreaded applications, the **_REENTRANT** flag must be defined on the command line at compilation time (**–D_REENTRANT**). When the **_REENTRANT** flag is defined, **errno** becomes a macro which enables each thread to have its own **errno.** This **errno** macro can be used on either side of the assignment , just as if it were a variable.

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

1 **EPERM** Not super-user
Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or the super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

2 **ENOENT** No such file or directory
A file name is specified and the file should exist but doesn't, or one of the directories in a path name does not exist.

3 **ESRCH** No such process, LWP, or thread
No process can be found in the system that corresponds to the specified PID, LWPID_t, or thread_t.

4 **EINTR** Interrupted system call
An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system service routine. If execution is resumed after processing the signal, it will appear as if the interrupted routine call returned this error condition.

In a multi-threaded application, linked with **libthread**, **EINTR** may be returned whenever another thread or LWP calls **fork**(2).

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

6 **ENXIO** No such device or address
I/O on a special file refers to a subdevice which does not exist, or exists beyond the limit of the device. It may also occur when, for example, a tape drive is not on-line or no disk pack is loaded on a drive.

7 **E2BIG** Arg list too long
An argument list longer than **ARG_MAX** bytes is presented to a member of the **exec** family of routines. The argument list limit is the sum of the size of the argument list plus the size of the environment's exported shell variables.

8 **ENOEXEC** Exec format error
A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid format (see **a.out**(4)).

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

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

11 **EAGAIN** No more processes, or no more LWPs
For example, the **fork** routine failed because the system's process table is full or the user is not allowed to create any more processes, or a system call failed because of insufficient memory or swap space.

12 **ENOMEM** Not enough space
During execution of an **exec**, **brk**, or **sbrk** routine, a program asks for more space than the system is able to supply. This is not a temporary condition; the maximum size is a system parameter. On some architectures, the error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers, or if there is not enough swap space during the **fork** routine. If this error occurs on a resource associated with Remote File Sharing (RFS), it indicates a memory depletion which may be temporary, dependent on system activity at the time the call was invoked.

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

14 **EFAULT** Bad address
The system encountered a hardware fault in attempting to use an argument of a routine. For example, **errno** potentially may be set to **EFAULT** any time a routine that takes a pointer argument is passed an invalid address, if the system can detect the condition. Because systems will differ in their ability to reliably detect a bad address, on some implementations passing a bad address to a routine will result in undefined behavior.

15 **ENOTBLK** Block device required
A non-block device or file was mentioned where a block device was required (for example, in a call to the **mount** routine).

16 **EBUSY** Device busy
An attempt was made to mount a device that was already mounted or an attempt was made to unmount a device on which there is an active file (open file, current directory, mounted-on file, active text segment). It will also occur if an attempt is made to enable accounting when it is already enabled. The device or resource is currently unavailable. **EBUSY** is also used by mutexs, semaphores, condition variables, and r / w locks, to indicate that a lock is held. And, **EBUSY** is also used by the processor control function **P_ONLINE**.

17 **EEXIST** File exists
An existing file was mentioned in an inappropriate context (for example, call to the **link** routine).

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

19 **ENODEV** No such device
An attempt was made to apply an inappropriate operation to a device (for example, read a write-only device).

20 **ENOTDIR** Not a directory
A non-directory was specified where a directory is required (for example, in a path prefix or as an argument to the **chdir** routine).

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

22 **EINVAL** Invalid argument
An invalid argument was specified (for example, unmounting a non-mounted device), mentioning an undefined signal in a call to the **signal** or **kill** routine.

23 **ENFILE** File table overflow
The system file table is full (that is, **SYS_OPEN** files are open, and temporarily no more files can be opened).

24 **EMFILE** Too many open files
No process may have more than **OPEN_MAX** file descriptors open at a time.

25 **ENOTTY** Inappropriate ioctl for device
A call was made to the **ioctl** routine specifying a file that is not a special character device.

26 **ETXTBSY** Text file busy (obsolete)
An attempt was made to execute a pure-procedure program that is currently open for writing. Also an attempt to open for writing or to remove a pure-procedure program that is being executed. *(This message is obsolete.)*

27 **EFBIG** File too large
The size of the file exceeded the limit specified by resource **RLIMIT_FSIZE**; or, the file size exceeds the maximum supported by the file system.

28 **ENOSPC** No space left on device
While writing an ordinary file or creating a directory entry, there is no free space left on the device. In the **fcntl** routine, the setting or removing of record locks on a file cannot be accomplished because there are no more record entries left on the system.

29 **ESPIPE** Illegal seek
A call to the **lseek** routine was issued to a pipe.

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

31 **EMLINK** Too many links
An attempt to make more than the maximum number of links, **LINK_MAX**, to a file.

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

33 **EDOM** Math argument out of domain of func
The argument of a function in the math package (3M) is out of the domain of the function.

34 **ERANGE** Math result not representable
The value of a function in the math package (3M) is not representable within machine precision.

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

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

37 **ECHRNG** Channel number out of range

38 **EL2NSYNC** Level 2 not synchronized

39 **EL3HLT** Level 3 halted

40 **EL3RST** Level 3 reset

41 **ELNRNG** Link number out of range

42 **EUNATCH** Protocol driver not attached

43 **ENOCSI** No CSI structure available

44 **EL2HLT** Level 2 halted

45 **EDEADLK** Deadlock condition
A deadlock situation was detected and avoided. This error pertains to file and record locking, and also applies to mutexs, semaphores, condition variables, and r / w locks.

46 **ENOLCK** No record locks available
There are no more locks available. The system lock table is full (see **fcntl**(2)).

47 **ECANCELED** Operation canceled
The associated asynchronous operation was canceled before completion.

48 **ENOTSUP** Not supported
This version of the system does not support this feature. Future versions of the system may provide support.

49 Reserved

58–59 Reserved

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

61 **ENODATA** No data available

62 **ETIME** Timer expired
The timer set for a STREAMS **ioctl** 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** operation is indeterminate. This is also returned in the case of **_lwp_cond_timedwait( )** or **cond_timedwait( ).**

63 **ENOSR** Out of stream resources
During a STREAMS **open**, either no STREAMS queues or no STREAMS head data structures were available. This is a temporary condition; one may recover from it if other processes release resources.

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

65 **ENOPKG** Package not installed
This error occurs when users attempt to use a system call from a package which has not been installed.

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

67 **ENOLINK** Link has been severed
This error is RFS specific. It occurs when the link (virtual circuit) connecting to a remote machine is gone.

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

69 **ESRMNT** Srmount error
This error is RFS specific. It occurs when an attempt is made to stop RFS while resources are still mounted by remote machines, or when a resource is readvertised with a client list that does not include a remote machine that currently has the resource mounted.

70 **ECOMM** Communication error on send
This error is RFS specific. It occurs when the current process is waiting for a message from a remote machine, and the virtual circuit fails.

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

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

76 **EDOTDOT** Error 76
This error is RFS specific. A way for the server to tell the client that a process has transferred back from mount point.

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

78 **ENAMETOOLONG** File name too long
The length of the path argument exceeds **PATH_MAX**, or the length of a path component exceeds **NAME_MAX** while **_POSIX_NO_TRUNC** is in effect; see **limits**(4).

79 **EOVERFLOW**
Value too large for defined data type.

80 **ENOTUNIQ** Name not unique on network
Given log name not unique.

81 **EBADFD** File descriptor in bad state
Either a file descriptor refers to no open file or a read request was made to a file that is open only for writing.

82 **EREMCHG** Remote address changed

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

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

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

86 **ELIBMAX** Attempting to link in more shared libraries than system limit
Trying to **exec** an **a.out** that requires more static shared libraries than is allowed on the current configuration of the system. See *NFS Administration Guide*.

87 **ELIBEXEC** Cannot **exec** a shared library directly
Attempting to **exec** a shared library directly.

88 **EILSEQ** Error 88
Illegal byte sequence. Handle multiple characters as a single character.

89 **ENOSYS** Operation not applicable

90 **ELOOP** Number of symbolic links encountered during path name traversal exceeds **MAXSYMLINKS**

91 **ESTART** Restartable system call
Interrupted system call should be restarted.

92 **ESTRPIPE** If pipe/FIFO, don't sleep in stream head
Streams pipe error (not externally visible).

93 **ENOTEMPTY** Directory not empty

94 **EUSERS** Too many users

95 **ENOTSOCK** Socket operation on non-socket

96 **EDESTADDRREQ** Destination address required
A required address was omitted from an operation on a transport endpoint. Destination address required.

97 **EMSGSIZE** Message too long
A message sent on a transport provider was larger than the internal message buffer or some other network limit.

98 **EPROTOTYPE** Protocol wrong type for socket
A protocol was specified that does not support the semantics of the socket type requested.

99 **ENOPROTOOPT** Protocol not available
A bad option or level was specified when getting or setting options for a protocol.

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

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

122 **EOPNOTSUPP** Operation not supported on transport endpoint
For example, trying to accept a connection on a datagram transport endpoint.

123 **EPFNOSUPPORT** Protocol family not supported
The protocol family has not been configured into the system or no implementation for it exists. Used for the Internet protocols.

124 **EAFNOSUPPORT** Address family not supported by protocol family
An address incompatible with the requested protocol was used.

125 **EADDRINUSE** Address already in use
User attempted to use an address already in use, and the protocol does not allow this.

126 **EADDRNOTAVAIL** Cannot assign requested address
Results from an attempt to create a transport endpoint with an address not on the current machine.

127 **ENETDOWN** Network is down
Operation encountered a dead network.

128 **ENETUNREACH** Network is unreachable
Operation was attempted to an unreachable network.

129 **ENETRESET** Network dropped connection because of reset
The host you were connected to crashed and rebooted.

130 **ECONNABORTED** Software caused connection abort
A connection abort was caused internal to your host machine.

131 **ECONNRESET** Connection reset by peer
A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote host due to a timeout or a reboot.

132 **ENOBUFS** No buffer space available
An operation on a transport endpoint or pipe was not performed because the system lacked sufficient buffer space or because a queue was full.

133 **EISCONN** Transport endpoint is already connected
A connect request was made on an already connected transport endpoint; or, a **sendto** or **sendmsg** request on a connected transport endpoint specified a destination when already connected.

134 **ENOTCONN** Transport endpoint is not connected
A request to send or receive data was disallowed because the transport endpoint is not connected and (when sending a datagram) no address was supplied.

143 **ESHUTDOWN** Cannot send after transport endpoint shutdown
A request to send data was disallowed because the transport endpoint has already been shut down.

144 **ETOOMANYREFS** Too many references: cannot splice

145 **ETIMEDOUT** Connection timed out
A connect or send request failed because the connected party did not properly respond after a period of time. (The timeout period is dependent on the communication protocol.)

146 **ECONNREFUSED** 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 remote host.

147 **EHOSTDOWN** Host is down
A transport provider operation failed because the destination host was down.

148 **EHOSTUNREACH** No route to host
A transport provider operation was attempted to an unreachable host.

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

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

151 **ESTALE** Stale NFS file handle

DEFINITIONS

Background Process Group Any process group that is not the foreground process group of a session that has established a connection with a controlling terminal.

Controlling Process A session leader that established a connection to a controlling terminal.

Controlling Terminal A terminal that is associated with a session. Each session may have, at most, one controlling terminal associated with it and a controlling terminal may be associated with only one session. Certain input sequences from the controlling terminal cause signals to be sent to process groups in the session associated with the controlling terminal; see **termio**(7I).

Directory Directories organize files into a hierarchical system where directories are the nodes in the hierarchy. A directory is a file that catalogues the list of files, including directories (subdirectories), that are directly beneath it in the hierarchy. Entries in a directory file are called links. A link associates a file identifier with a filename. By convention, a directory

contains at least two links, **.** (dot) and **..** (dot-dot). The link called dot refers to the directory itself while dot-dot refers to its parent directory. The root directory, which is the top-most node of the hierarchy, has itself as its parent directory. The pathname of the root directory is **/** and the parent directory of the root directory is **/**.

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

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

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

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

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

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

The effective user ID of the process does not match the user ID of the owner of the file, but either the effective group ID or one of the supplementary group IDs of the process match the group ID of the file and the appropriate access bit of the "group" portion (0070) of the file mode is set.

The effective user ID of the process does not match the user ID of the owner of the file, and neither the effective group ID nor any of the supplementary group IDs of the process match the group ID of the file, but the appropriate access bit of the "other" portion (0007) of the file mode is set.

Otherwise, the corresponding permissions are denied.

File Descriptor A file descriptor is a small integer used to do I / O on a file. The value of a file descriptor is from 0 to (**NOFILES**–**1**). A process may have no more than **NOFILES** file descriptors open simultaneously. A file descriptor is returned by system calls such as **open**, or **pipe**. The file descriptor is used as an argument by calls such as **read**, **write**, **ioctl**, and **close**.

File Name Names consisting of 1 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 character values excluding \0 (null) and the ASCII code for **/** (slash).

Note that it is generally unwise to use *, **?**, **[**, or **]** as part of file names because of the special meaning attached to these characters by the shell (see **sh**(1), **csh**(1), and **ksh**(1)). Although permitted, the use of unprintable characters in file names should be avoided.

A file name is sometimes referred to as a pathname component. The interpretation of a pathname component is dependent on the values of **NAME_MAX** and **_POSIX_NO_TRUNC** associated with the path prefix of that component. If any pathname component is longer than **NAME_MAX** and **_POSIX_NO_TRUNC** is in effect for the path prefix of that component (see **fpathconf**(2) and **limits**(4)), it shall be considered an error condition in that implementation. Otherwise, the implementation shall use the first **NAME_MAX** bytes of the pathname component.

Foreground Process Group

Each session that has established a connection with a controlling terminal will distinguish one process group of the session as the foreground process group of the controlling terminal. This group has certain privileges when accessing its controlling terminal that are denied to background process groups.

{IOV_MAX}

Maximum number of entries in a **struct iovec** array.

{LIMIT}

The braces notation, **{LIMIT}**, is used to denote a magnitude limitation imposed by the implementation. This indicates a value which may be defined by a header file (without the braces), or the actual value may be obtained at runtime by a call to the configuration inquiry **pathconf**(2) with the name argument **_PC_LIMIT**.

Masks

The file mode creation mask of the process used during any create function calls to turn off permission bits in the *mode* argument supplied. Bit positions that are set in **umask(***cmask***)** are cleared in the mode of the created file.

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.

Message Queue Identifier

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

```
struct    ipc_perm msg_perm;
struct    msg *msg_first;
struct    msg *msg_last;
ulong     msg_cbytes;
ulong     msg_qnum;
ulong     msg_qbytes;
pid_t     msg_lspid;
pid_t     msg_lrpid;
time_t    msg_stime;
```

```
time_t      msg_rtime;
time_t      msg_ctime;
```

Here are descriptions of the fields of the **msqid_ds** structure:

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

```
uid_t       cuid;     /* creator user id */
gid_t       cgid;     /* creator group id */
uid_t       uid;      /* user id */
gid_t       gid;      /* group id */
mode_t      mode;     /* r/w permission */
ulong       seq;      /* slot usage sequence # */
key_t       key;      /* key */
```

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

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

msg_cbytes is the current number of bytes on the queue.

msg_qnum is the number of messages currently on the queue.

msg_qbytes is the maximum number of bytes allowed on the queue.

msg_lspid is the process ID of the last process that performed a **msgsnd** operation.

msg_lrpid is the process id of the last process that performed a **msgrcv** operation.

msg_stime is the time of the last **msgsnd** operation.

msg_rtime is the time of the last **msgrcv** operation

msg_ctime is the time of the last **msgctl** operation that changed a member of the above structure.

Message Operation Permissions

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

```
00400     READ by user
00200     WRITE by user
00040     READ by group
00020     WRITE by group
00004     READ by others
00002     WRITE by others
```

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

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

The effective user ID of the process matches **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with **msqid** and the appropriate bit of the “user”

portion (0600) of **msg_perm.mode** is set.

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

The appropriate bit of the "other" portion (006) of **msg_perm.mode** is set.

Otherwise, the corresponding permissions are denied.

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.

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.

Orphaned Process Group

A process group in which the parent of every member in the group is either itself a member of the group, or is not a member of the process group's session.

Path Name

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

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

A slash by itself names the root directory.

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

Process ID

Each process in the system is uniquely identified during its lifetime by a positive integer called a process ID. A process ID may not be reused by the system until the process lifetime, process group lifetime and session lifetime ends for any process ID, process group ID and session ID equal to that process ID. Within a processs, there are threads with thread id's, called thread_t and LWPID_t. These threads are not visable to the outside process.

Parent Process ID

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

Privilege

Having appropriate privilege means having the capability to override system restrictions.

Process Group

Each process in the system is a member of a process group that is identified by a process group ID. Any process that is not a process group leader may create a new process group and become its leader. Any process that is not a process group leader may join an

existing process group that shares the same session as the process. A newly created process joins the process group of its parent.

Process Group Leader

A process group leader is a process whose process ID is the same as its process group ID.

Process Group ID

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

Process Lifetime

A process lifetime begins when the process is forked and ends after it exits, when its termination has been acknowledged by its parent process. See **wait**(2).

Process Group Lifetime

A process group lifetime begins when the process group is created by its process group leader, and ends when the lifetime of the last process in the group ends or when the last process in the group leaves the group.

Read Queue

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

Real User ID and Real Group ID

Each user allowed on the system is identified by a positive integer (0 to **MAXUID**) called a real user ID.

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

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

Root Directory and Current Working Directory

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

Saved User ID and Saved Group ID

The saved user ID and saved group ID are the values of the effective user ID and effective group ID prior to an exec of a file whose set user or set group file mode bit has been set (see **exec**(2)).

Semaphore Identifier

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

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

Here are descriptions of the fields of the **semid_ds** structure:

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

```
uid_t   uid;    /* user id */
gid_t   gid;    /* group id */
uid_t   cuid;   /* creator user id */
gid_t   cgid;   /* creator group id */
mode_t  mode;   /* r/a permission */
ulong   seq;    /* slot usage sequence number */
key_t   key;    /* key */
```

sem_nsems is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a nonnegative 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** operation.

sem_ctime is the time of the last **semctl** operation that changed a member of the above structure.

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

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

semval is a non-negative integer that is the actual value of the semaphore.

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 0.

Semaphore Operation Permissions

In the **semop** and **semctl** 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
00040   READ by group
00020   ALTER by group
00004   READ by others
00002   ALTER by others
```

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

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

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

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

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

Otherwise, the corresponding permissions are denied.

Session

A session is a group of processes identified by a common ID called a session ID, capable of establishing a connection with a controlling terminal. Any process that is not a process group leader may create a new session and process group, becoming the session leader of the session and process group leader of the process group. A newly created process joins the session of its creator.

Session ID

Each session in the system is uniquely identified during its lifetime by a positive integer called a session ID, the process ID of its session leader.

Session Leader

A session leader is a process whose session ID is the same as its process and process group ID.

Session Lifetime

A session lifetime begins when the session is created by its session leader, and ends when the lifetime of the last process that is a member of the session ends, or when the last process that is a member in the session leaves the session.

Shared Memory Identifier

A shared memory identifier (**shmid**) is a unique positive integer created by a **shmget** system call. Each **shmid** has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. (Note that these shared memory segments must be explicitly removed by the user after the last reference to them is removed.) The data structure is referred to as **shmid_ds** and contains the following members:

```
struct ipc_perm   shm_perm;    /* operation permission struct */
int               shm_segsz;   /* size of segment */
struct region     *shm_reg;    /* ptr to region structure */
char              pad[4];      /* for swap compatibility */
pid_t             shm_lpid;    /* pid of last operation */
pid_t             shm_cpid;    /* creator pid */
ushort            shm_nattch;  /* number of current attaches */
ushort            shm_cnattch; /* used only for shminfo */
time_t            shm_atime;   /* last attach time */
time_t            shm_dtime;   /* last detach time */
time_t            shm_ctime;   /* last change time */
                               /* Times measured in secs since */
                               /* 00:00:00 GMT, Jan. 1, 1970 */
```

Here are descriptions of the fields of the **shmid_ds** structure:

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

```
uid_t   cuid;   /* creator user id */
gid_t   cgid;   /* creator group id */
uid_t   uid;    /* user id */
gid_t   gid;    /* group id */
mode_t  mode;   /* r/w permission */
ulong   seq;    /* slot usage sequence # */
key_t   key;    /* key */
```

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

shm_cpid is the process ID of the process that created the shared memory identifier.

shm_lpid is the process ID of the last process that performed a **shmop** operation.

shm_nattch is the number of processes that currently have this segment attached.

shm_atime is the time of the last **shmat** operation (see **shmop**(2)).

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

shm_ctime is the time of the last **shmctl** operation that changed one of the members of the above structure.

Shared Memory Operation Permissions

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

00400 READ by user
00200 WRITE by user
00040 READ by group
00020 WRITE by group
00004 READ by others
00002 WRITE by others

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

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

The effective user ID of the process matches **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with **shmid** and the appropriate bit of the “user” portion (0600) of **shm_perm.mode** is set.

The effective group ID of the process matches **shm_perm.cgid** or **shm_perm.gid** and the appropriate bit of the “group” portion (060) of **shm_perm.mode** is set.

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

Otherwise, the corresponding permissions are denied.

Special Processes The process with ID 0 and the process with ID 1 are special processes referred to as proc0 and proc1; see **kill**(2). proc0 is the process scheduler. proc1 is the initialization process (**init**); proc1 is the ancestor of every other process in the system and is used to control the process structure.

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.

Super-user A process is recognized as a super-user process and is granted special privileges, such as immunity from file permissions, if its effective user ID is 0.

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

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

Name	Description
access(2)	determine accessibility of a file
acct(2)	enable or disable process accounting
acl(2)	get or set a file's Access Control List (ACL)
adjtime(2)	correct the time to allow synchronization of the system clock
alarm(2)	set a process alarm clock
audit(2)	write a record to the audit log
auditon(2)	manipulate auditing
auditsvc(2)	write audit log to specified file descriptor
brk(2)	change the amount of space allocated for the calling process's data segment
chdir(2)	change working directory

chmod(2)	change access permission mode of file
chown(2)	change owner and group of a file
chroot(2)	change root directory
close(2)	close a file descriptor
creat(2)	create a new file or rewrite an existing one
door(2)	Solaris 2.5 internal implementation detail
door_call(2)	See **door**(2)
door_create(2)	See **door**(2)
door_info(2)	See **door**(2)
door_return(2)	See **door**(2)
door_revoke(2)	See **door**(2)
dup(2)	duplicate an open file descriptor
exec(2)	execute a file
execl(2)	See **exec**(2)
execle(2)	See **exec**(2)
execlp(2)	See **exec**(2)
execv(2)	See **exec**(2)
execve(2)	See **exec**(2)
execvp(2)	See **exec**(2)
_exit(2)	See **exit**(2)
exit(2)	terminate process
facl(2)	See **acl**(2)
fchdir(2)	See **chdir**(2)
fchmod(2)	See **chmod**(2)
fchown(2)	See **chown**(2)
fchroot(2)	See **chroot**(2)
fcntl(2)	file control
fork(2)	create a new process
fork1(2)	See **fork**(2)
fpathconf(2)	get configurable pathname variables
fstat(2)	See **stat**(2)
fstatvfs(2)	See **statvfs**(2)
getaudit(2)	get and set process audit information
getauid(2)	get and set user audit identity
getcontext(2)	get and set current user context

getdents(2)	read directory entries and put in a file system independent format
getegid(2)	See **getuid**(2)
geteuid(2)	See **getuid**(2)
getgid(2)	See **getuid**(2)
getgroups(2)	get or set supplementary group access list IDs
getitimer(2)	get or set value of interval timer
getmsg(2)	get next message off a stream
getpgid(2)	See **getpid**(2)
getpgrp(2)	See **getpid**(2)
getpid(2)	get process, process group, and parent process IDs
getpmsg(2)	See **getmsg**(2)
getppid(2)	See **getpid**(2)
getrlimit(2)	control maximum system resource consumption
getsid(2)	get or set session ID
getuid(2)	get real user, effective user, real group, and effective group IDs
ioctl(2)	control device
kill(2)	send a signal to a process or a group of processes
lchown(2)	See **chown**(2)
link(2)	link to a file
llseek(2)	move extended read / write file pointer
lseek(2)	move read / write file pointer
lstat(2)	See **stat**(2)
_lwp_cond_broadcast(2)	See **_lwp_cond_signal**(2)
_lwp_cond_signal(2)	signal a condition variable
_lwp_cond_timedwait(2)	See **_lwp_cond_wait**(2)
_lwp_cond_wait(2)	wait on a condition variable
_lwp_continue(2)	See **_lwp_suspend**(2)
_lwp_create(2)	create a new light-weight process
_lwp_exit(2)	terminate the calling LWP
_lwp_getprivate(2)	See **_lwp_setprivate**(2)
_lwp_info(2)	return the time-accounting information of a single LWP
_lwp_kill(2)	send a signal to a LWP
_lwp_makecontext(2)	initialize an LWP context
_lwp_mutex_lock(2)	mutual exclusion

_lwp_mutex_trylock(2)	See **_lwp_mutex_lock**(2)
_lwp_mutex_unlock(2)	See **_lwp_mutex_lock**(2)
_lwp_self(2)	get LWP identifier
_lwp_sema_init(2)	See **_lwp_sema_wait**(2)
_lwp_sema_post(2)	See **_lwp_sema_wait**(2)
_lwp_sema_wait(2)	semaphore operations
_lwp_setprivate(2)	set / get LWP specific storage
_lwp_sigredirect(2)	See **_signotifywait**(2)
_lwp_suspend(2)	continue or suspend LWP execution
_lwp_wait(2)	wait for a LWP to terminate
memcntl(2)	memory management control
mincore(2)	determine residency of memory pages
mkdir(2)	make a directory
mknod(2)	make a directory, or a special or ordinary file
mmap(2)	map pages of memory
mount(2)	mount a file system
mprotect(2)	set protection of memory mapping
msgctl(2)	message control operations
msgget(2)	get message queue
msgop(2)	message operations
msgrcv(2)	See **msgop**(2)
msgsnd(2)	See **msgop**(2)
munmap(2)	unmap pages of memory
nice(2)	change priority of a process
open(2)	open for reading or writing
pathconf(2)	See **fpathconf**(2)
pause(2)	suspend process until signal
pipe(2)	create an interprocess channel
poll(2)	input / output multiplexing
p_online(2)	change processor online or offline status
pread(2)	See **read**(2)
priocntl(2)	process scheduler control
priocntlset(2)	generalized process scheduler control
processor_bind(2)	bind LWPs to a processor
processor_info(2)	determine type and status of a processor

profil(2)	execution time profile
ptrace(2)	allows a parent process to control the execution of a child process
putmsg(2)	send a message on a stream
putpmsg(2)	See **putmsg**(2)
pwrite(2)	See **write**(2)
read(2)	read from file
readlink(2)	read the value of a symbolic link
readv(2)	See **read**(2)
rename(2)	change the name of a file
rmdir(2)	remove a directory
sbrk(2)	See **brk**(2)
semctl(2)	semaphore control operations
semget(2)	get set of semaphores
semop(2)	semaphore operations
setaudit(2)	See **getaudit**(2)
setauid(2)	See **getauid**(2)
setcontext(2)	See **getcontext**(2)
setegid(2)	See **setuid**(2)
seteuid(2)	See **setuid**(2)
setgid(2)	See **setuid**(2)
setgroups(2)	See **getgroups**(2)
setitimer(2)	See **getitimer**(2)
setpgid(2)	set process group ID
setpgrp(2)	set process group ID
setregid(2)	set real and effective group IDs
setreuid(2)	set real and effective user IDs
setrlimit(2)	See **getrlimit**(2)
setsid(2)	See **getsid**(2)
setuid(2)	set user and group IDs
shmat(2)	See **shmop**(2)
shmctl(2)	shared memory control operations
shmdt(2)	See **shmop**(2)
shmget(2)	get shared memory segment identifier
shmop(2)	shared memory operations

sigaction(2)	detailed signal management
sigaltstack(2)	set or get signal alternate stack context
_signotifywait(2)	deliver process signals to specific LWPs
sigpending(2)	examine signals that are blocked and pending
sigprocmask(2)	change and / or examine calling process's signal mask
sigsend(2)	send a signal to a process or a group of processes
sigsendset(2)	See **sigsend**(2)
sigsuspend(2)	install a signal mask and suspend process until signal
sigwait(2)	wait until a signal is posted
stat(2)	get file status
statvfs(2)	get file system information
stime(2)	set system time and date
swapctl(2)	manage swap space
symlink(2)	make a symbolic link to a file
sync(2)	update super block
sysfs(2)	get file system type information
sysinfo(2)	get and set system information strings
time(2)	get time
times(2)	get process and child process times
uadmin(2)	administrative control
ulimit(2)	get and set process limits
umask(2)	set and get file creation mask
umount(2)	unmount a file system
uname(2)	get name of current operating system
unlink(2)	remove directory entry
ustat(2)	get file system statistics
utime(2)	set file access and modification times
utimes(2)	set file times
vfork(2)	spawn new process in a virtual memory efficient way
vhangup(2)	virtually "hangup" the current controlling terminal
wait(2)	wait for child process to stop or terminate
waitid(2)	wait for child process to change state
waitpid(2)	wait for child process to change state
write(2)	write on a file
writev(2)	See **write**(2)

yield(2)	yield execution to another lightweight process

NAME

_lwp_cond_signal, _lwp_cond_broadcast – signal a condition variable

SYNOPSIS

```
#include <sys/lwp.h>
int _lwp_cond_signal(lwp_cond_t cvp);
int _lwp_cond_broadcast(lwp_cond_t *cvp);
```

DESCRIPTION

_lwp_cond_signal( ) unblocks one LWP that is blocked on the LWP condition variable pointed to by *cvp*.

_lwp_cond_broadcast( ) unblocks all LWPs that are blocked on the LWP condition variable pointed to by *cvp*.

If no LWPs are blocked on the LWP condition variable, then **_lwp_cond_signal( )** and **_lwp_cond_broadcast( )** have no effect.

Both functions should be called under the protection of the same LWP mutex lock that is used with the LWP condition variable being signalled. Otherwise the condition variable may be signalled between the test of the associated condition and blocking in **_lwp_cond_wait( )**. This can cause an infinite wait.

RETURN VALUES

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, **_lwp_cond_signal( )**, and **_lwp_cond_broadcast( )** fail and return the corresponding value:

EINVAL *cvp* points to an invalid LWP condition variable.

EFAULT *cvp* points to an invalid address.

SEE ALSO

_lwp_cond_wait(2), **_lwp_mutex_lock**(2)

NAME

_lwp_cond_wait, _lwp_cond_timedwait – wait on a condition variable

SYNOPSIS

```
#include <sys/lwp.h>
int _lwp_cond_wait(lwp_cond_t *cvp, lwp_mutex_t *mp);
int _lwp_cond_timedwait(lwp_cond_t *cvp, lwp_mutex_t *mp, timestruc_t *abstime);
```

DESCRIPTION

These functions are used to wait for the occurrence of a condition represented by an LWP condition variable. LWP condition variables must be initialized to zero before use.

_lwp_cond_wait( ) atomically releases the LWP mutex pointed to by *mp* and causes the calling LWP to block on the LWP condition variable pointed to by *cvp*. The blocked LWP may be awakened by **_lwp_cond_signal**(2), **_lwp_cond_broadcast**(2), or when interrupted by delivery of a signal. Any change in value of a condition associated with the condition variable cannot be inferred by the return of **_lwp_cond_wait( )** and any such condition must be re-evaluated.

_lwp_cond_timedwait( ) is similar to **_lwp_cond_wait( )**, except that the calling LWP will not block past the time of day specified by *abstime*. If the time of day becomes greater than *abstime* then **_lwp_cond_timedwait( )** returns with the error code **ETIME .**

_lwp_cond_wait( ), and**_lwp_cond_timedwait( )** always return with the mutex locked and owned by the calling lightweight process.

RETURN VALUES

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, **_lwp_cond_wait( )**, and **_lwp_cond_timedwait( )** fail and return the corresponding value:

EINVAL *cvp* points to an invalid LWP condition variable or *mp* points to an invalid LWP mutex.

EFAULT *mp*, *cvp*, or *abstime* point to an illegal address.

If any of the following conditions occur, **_lwp_cond_wait( )**, and **_lwp_cond_timedwait( )** fail and return the corresponding value:

EINTR The call was interrupted by a signal or **fork**(2).

If any of the following conditions occur, **_lwp_cond_timedwait( )** fails and returns the corresponding value:

ETIME The time specified in *abstime* has passed.

EXAMPLES

_lwp_cond_wait() is normally used in a loop testing some condition, as follows:

```
lwp_mutex_t m;
lwp_cond_t cv;
int cond;

(void) _lwp_mutex_lock(&m);
while (cond == FALSE) {
        (void) _lwp_cond_wait(&cv, &m);
}
(void) _lwp_mutex_unlock(&m);
```

_lwp_cond_timedwait() is also normally used in a loop testing some condition. It uses an absolute timeout value as follows:

```
timestruc_t to;
lwp_mutex_t m;
lwp_cond_t cv;
int cond, err;

(void) _lwp_mutex_lock(&m);
to.tv_sec = time(NULL) + TIMEOUT;
to.tv_nsec = 0;
while (cond == FALSE) {
        err = _lwp_cond_timedwait(&cv, &m, &to);
        if (err == ETIME) {
                /* timeout, do something */
                break;
        }
}
(void) _lwp_mutex_unlock(&m);
```

This sets a bound on the total wait time even though the **_lwp_cond_timedwait()** may return several times due to the condition being signalled or the wait being interrupted.

SEE ALSO

_lwp_cond_broadcast(2), **_lwp_cond_signal**(2), **_lwp_kill**(2), **_lwp_mutex_lock**(2), **fork**(2), **kill**(2)

NAME

_lwp_create – create a new light-weight process

SYNOPSIS

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

int _lwp_create(ucontext_t *contextp, unsigned long flags, lwpid_t *new_lwp);
```

DESCRIPTION

The function **_lwp_create( )** adds a lightweight process (LWP) to the current process. The *context* parameter specifies the initial signal mask, stack, and machine context (including the program counter and stack pointer) for the new LWP. The new LWP inherits the scheduling class and priority of the caller.

If **_lwp_create( )** is successful, the ID of the new LWP is stored in the location pointed to by *new_lwp*.

flags specifies additional attributes for the new LWP. The value in *flags* is constructed by the bit-wise inclusive OR of the following values:

LWP_DETACHED	The LWP is created detached.
LWP_SUSPENDED	The LWP is created suspended. If **__LWP_ASLWP** is specified, then the LWP created is the special, designated LWP which handles signals sent to a multi-threaded process — the **aslwp.** There can be only one **aslwp** in a multi-threaded process, so the creation of another **aslwp** will return an error code — **EINVAL.**
__LWP_ASLWP	The LWP created is the **aslwp** (Asynchronous Signals LWP) (see **signal**(5)). The **aslwp** should always have all signals blocked — that is how it should be created. It should never exit via **_lwp_exit**(2) or **exit**(2). This flag should not be used by any user program. It is documented here purely for the sake of documentation and not for use by an application.

If **LWP_DETACHED** is specified, then the LWP is created in the *detached* state. Otherwise the LWP is created in the undetached state. The ID (and system resources) associated with a detached LWP can be automatically reclaimed when the LWP exits. The ID of an undetached LWP cannot be reclaimed until it exits and another LWP has reported its termination via **_lwp_wait**(2). This allows the waiting LWP to determine that the waited for LWP has terminated and to reclaim any process resources that it was using.

If **LWP_SUSPENDED** is specified, then the LWP is created in a suspended state. This allows the creator to change the LWP's inherited attributes before it starts to execute. The suspended LWP can only be resumed via **_lwp_continue**(2). If **LWP_SUSPENDED** is not specified the LWP can begin to run immediately after it has been created.

RETURN VALUES

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, **_lwp_create( )** fails and returns the corresponding value:

EFAULT	Either the *context* parameter or the *new_lwp* parameter point to invalid addresses.
EAGAIN	A system limit is exceeded, e.g., too many LWP were created for this real user ID.
EINVAL	The **__LWP_ASLWP** flag was used to create more than one **aslwp** in the process. *There can be only one aslwp within a process.*

EXAMPLES

This example shows how a stack is allocated to a new LWP. **_lwp_makecontext( )** is used to set up the *context* parameter so that the new LWP begins executing a function.

```
contextp = (ucontext_t *)malloc(sizeof(ucontext_t));
stackbase = malloc(stacksize);
sigprocmask(SIGSETMASK, NULL, &contextp->uc_sigmask);
_lwp_makecontext(contextp, func, arg, private, stackbase, stacksize);
error = _lwp_create(contextp, NULL, &new_lwp);
```

SEE ALSO

_lwp_continue(2), **_lwp_exit**(2), **_lwp_makecontext**(2), **_lwp_wait**(2), **exit**(2), **signal**(5), **ucontext**(5)

NAME

_lwp_exit – terminate the calling LWP

SYNOPSIS

```
#include <sys/lwp.h>
void _lwp_exit(void);
```

DESCRIPTION

_lwp_exit() causes the calling LWP to terminate. If it is the last LWP in the process, then the process exits with a status of zero (see **exit**(2).

If the LWP was created undetached, it is transformed into a "zombie LWP" that retains at least the LWP's ID until it is waited for (see **_lwp_wait**(2)). Otherwise, its ID and system resources may be reclaimed immediately.

SEE ALSO

_lwp_create(2), **_lwp_wait**(2), **exit**(2)

NAME

_lwp_info – return the time-accounting information of a single LWP

SYNOPSIS

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

int _lwp_info(struct lwpinfo *buffer);
```

DESCRIPTION

_lwp_info() fills the **lwpinfo** structure pointed to by *buffer* with time-accounting information pertaining to the calling LWP. This call may be extended in the future to return other information to the **lwpinfo** structure as needed. The **lwpinfo** structure in **<sys/lwp.h>** includes the following members:

```
timestruc_t lwp_utime;
timestruc_t lwp_stime;
```

lwp_utime is the CPU time used while executing instructions in the user space of the calling LWP.

lwp_stime is the CPU time used by the system on behalf of the calling LWP.

RETURN VALUES

Upon successful completion, **_lwp_info()** returns **0** and fills in the **lwpinfo** structure pointed to by *buffer*.

ERRORS

If the following condition is detected, **_lwp_info()** returns the corresponding value:

EFAULT *buffer* points to an illegal address.

SEE ALSO

times(2)

NAME

_lwp_kill – send a signal to a LWP

SYNOPSIS

```
#include <sys/lwp.h>
#include <signal.h>

int _lwp_kill(lwpid_t target_lwp, int sig);
```

DESCRIPTION

_lwp_kill() sends a signal to the LWP specified by *target_lwp*. The signal that is to be sent is specified by *sig* and must be one from the list given in **signal**(5). If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *target_lwp*.

The *target_lwp* must be an LWP within the same process as the calling LWP.

RETURN VALUES

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions occur, **_lwp_kill()** fails and returns the corresponding value:

EINVAL *sig* is not a valid signal number.

ESRCH *target_lwp* cannot be found in the current process.

SEE ALSO

kill(2), **sigaction**(2), **sigprocmask**(2), **signal**(5)

NAME

_lwp_makecontext – initialize an LWP context

SYNOPSIS

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

void _lwp_makecontext(ucontext_t *ucp, void (*start_routine)( void *), void *arg,
        void *private, caddr_t stack_base, size_t stack_size);
```

DESCRIPTION

_lwp_makecontext( ) initializes the user context structure pointed to by *ucp*. The user context is defined by **ucontext**(5). The resulting user context can be used by **_lwp_create**(2) for specifying the initial state of the new LWP. The user context is set up to start executing the function *start_routine* with a single argument, *arg*, and to call **_lwp_exit**(2) if *start_routine* returns. The new LWP will use the storage starting at *stack_base* and continuing for *stack_size* bytes as an execution stack. The initial value in LWP-private memory will be set to *private* (see **_lwp_setprivate**(2)). The signal mask in the user context is not initialized.

SEE ALSO

_lwp_create(2), **_lwp_exit**(2), **_lwp_setprivate**(2), **ucontext**(5)

NAME

_lwp_mutex_lock, _lwp_mutex_unlock, _lwp_mutex_trylock – mutual exclusion

SYNOPSIS

```
#include <sys/lwp.h>
int _lwp_mutex_lock(lwp_mutex_t *mp);
int _lwp_mutex_trylock(lwp_mutex_t *mp);
int _lwp_mutex_unlock(lwp_mutex_t *mp);
```

DESCRIPTION

These functions serialize the execution of lightweight processes. They are useful for ensuring that only one lightweight process can execute a critical section of code at any one time (mutual exclusion). LWP mutexes must be initialized to zero before use.

_lwp_mutex_lock( ) locks the LWP mutex pointed to by *mp*. If the mutex is already locked, the calling LWP blocks until the mutex becomes available. When **_lwp_mutex_lock( )** returns, the mutex is locked and the calling LWP is the "owner".

_lwp_mutex_trylock( ) attempts to lock the mutex. If the mutex is already locked it returns with an error. If the mutex is unlocked, it is locked and **_lwp_mutex_trylock( )** returns.

_lwp_mutex_unlock( ) unlocks a locked mutex. The mutex must be locked and the calling LWP must be the one that last locked the mutex (the owner). If any other LWPs are waiting for the mutex to become available, one of them is unblocked.

RETURN VALUES

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, **_lwp_mutex_lock( )**, **_lwp_mutex_trylock( )**, and **_lwp_mutex_unlock( )** fail and return the corresponding value:

EINVAL *mp* points to an invalid LWP mutex.

EFAULT *mp* points to an illegal address.

If any of the following conditions occur, **_lwp_mutex_trylock( )** fails and returns the corresponding value:

EBUSY *mp* points to a locked mutex.

SEE ALSO

intro(2), **_lwp_cond_wait**(2)

NAME

_lwp_self – get LWP identifier

SYNOPSIS

```
#include <sys/lwp.h>
lwpid_t _lwp_self(void);
```

DESCRIPTION

_lwp_self() returns the ID of the calling LWP.

SEE ALSO

_lwp_create(2)

NAME

_lwp_sema_wait, _lwp_sema_init, _lwp_sema_post – semaphore operations

SYNOPSIS

```
#include <sys/lwp.h>
int _lwp_sema_wait(lwp_sema_t *sema);
int _lwp_sema_init(lwp_sema_t *sema, int count);
int _lwp_sema_post(lwp_sema_t *sema);
```

DESCRIPTION

Conceptually, a semaphore is an non-negative integer count that is atomically incremented and decremented. Typically this represents the number of resources available. **_lwp_sema_init( )** initializes the count, **_lwp_sema_post( )** atomically increments the count, and **_lwp_sema_wait( )** waits for the count to become greater than zero and then atomically decrements it.

LWP semaphores must be initialized before use. **_lwp_sema_init( )** initializes the count associated with the LWP semaphore pointed to by *sema* to *count*.

_lwp_sema_wait( ) blocks the calling LWP until the semaphore count becomes greater than zero and then atomically decrements it.

_lwp_sema_post( ) atomically increments the semaphore count. If there are any LWPs blocked on the semaphore, one is unblocked.

RETURN VALUES

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, **_lwp_sema_init( )**, **_lwp_sema_wait( )**, and **_lwp_sema_post( )** fail and return the corresponding value:

EINVAL	*sema* points to an invalid semaphore.
EFAULT	*sema* points to an illegal address.
EINTR	**_lwp_sema_wait( )** was interrupted by a signal or **fork**(2).

SEE ALSO

fork(2)

NAME

_lwp_setprivate, _lwp_getprivate – set / get LWP specific storage

SYNOPSIS

#include <sys/lwp.h>

void _lwp_setprivate(void ∗*buffer*);

void ∗**_lwp_getprivate(void);**

DESCRIPTION

The function **_lwp_setprivate( )** stores the value specified by *buffer* in LWP-private memory that is unique to the calling LWP. This is typically used by thread library implementations to maintain a pointer to information about the thread currently running on the calling LWP.

The function **_lwp_getprivate( )** returns the value stored in LWP-private memory.

SEE ALSO

_lwp_makecontext(2)

NAME

_lwp_suspend, _lwp_continue – continue or suspend LWP execution

SYNOPSIS

```
#include <sys/lwp.h>
int _lwp_suspend(lwpid_t target_lwp);
int _lwp_continue(lwpid_t target_lwp);
```

DESCRIPTION

_lwp_suspend( ) immediately suspends the execution of the LWP specified by *target_lwp*. On successful return from **_lwp_suspend( )**, *target_lwp* is no longer executing. Once a thread is suspended, subsequent calls to **_lwp_suspend( )** have no affect.

_lwp_continue( ) resumes the execution of a suspended LWP. Once a suspended LWP is continued, subsequent calls to **_lwp_continue( )** have no effect.

A suspended LWP will not be awakened by a signal. The signal stays pending until the execution of the LWP is resumed by **_lwp_continue( )**.

RETURN VALUES

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If the following condition occurs, **_lwp_suspend( )** and **_lwp_continue( )** fail and return the corresponding value:

ESRCH *target_lwpid* cannot be found in the current process

If the following condition is detected, **_lwp_suspend( )** fails and returns the corresponding value:

EDEADLK Suspending *target_lwpid* will cause all LWPs in the process to be suspended.

SEE ALSO

_lwp_create(2)

NAME

_lwp_wait – wait for a LWP to terminate

SYNOPSIS

```
#include <sys/lwp.h>
int _lwp_wait(lwpid_t wait_for, lwpid_t *departed_lwp);
```

DESCRIPTION

_lwp_wait( ) blocks the current LWP until the LWP specified by *wait_for* terminates. If the specified LWP terminated prior to the call to **_lwp_wait( )**, then **_lwp_wait( )** returns immediately. If *wait_for* is **NULL**, then **_lwp_wait( )** waits for any undetached LWP in the current process. If *wait_for* is not **NULL**, then it must specify an undetached LWP in the current process. If *departed_lwp* is not **NULL**, then it points to location where the ID of the exited LWP is stored (see **_lwp_exit**(2)).

When an LWP exits and there are one or more LWPs in this process waiting for this specific LWP to exit, then one of the waiting LWPs is unblocked and it returns from **_lwp_wait( )** successfully. Any other LWPs waiting for this same LWP to exit are also unblocked, however, they return from **_lwp_wait( )** with an error (**ESRCH**) indicating the waited for LWP no longer exists. If there are no LWPs in this process waiting for this specific LWP to exit but there are one or more LWPs waiting for any LWP to exit, then one of the waiting LWPs is unblocked and it returns from **_lwp_wait( )** successfully.

The ID of an LWP that has exited may be reused via **_lwp_create( )** after the LWP has been successfully waited for.

RETURN VALUES

Zero is returned when successful. A non-zero value indicates an error.

ERRORS

If any of the following conditions are detected, **_lwp_wait( )** fails and returns the corresponding value:

EINTR	**_lwp_wait( )** was interrupted by a signal.
EDEADLK	All LWPs in this process would be blocked waiting for LWPs to terminate.
EDEADLK	The calling LWP is attempting to wait for itself.

If any of the following conditions occur, **_lwp_wait( )** fails and returns the corresponding value:

ESRCH	*wait_for* cannot be found in the current process or it was detached.

SEE ALSO

_lwp_create(2), **_lwp_exit**(2)

NAME

_signotifywait, _lwp_sigredirect – deliver process signals to specific LWPs

SYNOPSIS

```
#include <sys/lwp.h>
int _signotifywait(void);
int _lwp_sigredirect(lwpid_t target_lwp, int signo);
```

DESCRIPTION

In a multi-threaded process, signals that are generated for a process are delivered to one of the threads that does not have that signal masked. If all of the application threads are masking that signal, its delivery waits until one of them unmasks it.

The disposition of the each thread's signal mask is unknown to the kernel when it generates signals for the process. The **_signotifywait( )** and **_lwp_sigredirect( )** functions provide a mechanism to direct instances of signals generated for the process to application-specified LWPs. Each process has a set of signals pending for the process, and for each LWP there is a set of signals pending for that LWP. If no signals are pending, these sets are empty.

There is also a process-wide signal set, termed the *notification* set, manipulated by these functions. A signal generated for the process where the signal number is not in the notification set is called an *unnotified* signal.

In a multi-threaded program there is an **aslwp,** a special LWP endowed with powers to handle signals that are generated for a process. The **_signotifywait( )** function is used to await signals generated for the process, and should be called only from the **aslwp.** In general, these functions are not to be called from the application-level.

If there is a pending unnotified signal when **_signotifywait( )** is called, that signal is selected and the call returns immediately. If there is not a signal pending, the call suspends the calling LWP until the generation of an unnotified signal; that signal then is selected and the function returns. In both cases, the selected signal number is set in the notification set and returned as the value of **_signotifywait( )**. The signal remains pending for the process, and any associated **siginfo**(5) information remains queued at the process.

The **_lwp_sigredirect( )** function requests that a signal pending for the process be delivered to the LWP specified by *target_lwp*. It is an error if *signo* is not currently in the notification set of the process. The signal specified by *signo* is removed from pending for the process and is made pending for the *target_lwp*. If there is an associated *siginfo* information structure queued at the process, that *siginfo* is queued to the *target_lwp*.

Whenever a signal is cleared from the set of signals pending for the process, the corresponding signal is cleared from the notification set. After a successful call to **_lwp_sigredirect( )**, the signal *signo* is cleared from the notification set and from the set of signals pending for the process. If another instance of *signo* is queued for the process, the signal number is again set in the process pending mask, and if another LWP is blocked in a call to **_signotifywait( )**, its wait for an unnotified signal will be satisfied. The effects described in this paragraph also apply when the signal *signo* is returned by a call to **sigtimedwait( )** and *signo* was not pending for the calling LWP.

RETURN VALUES

The function **_signotifywait( )** returns the signal number of the pending but hitherto unnotified signal. The function **_lwp_sigredirect( )** returns zero when successful; otherwise, a non-zero value indicates an error.

ERRORS

No error conditions are specified for **_signotifywait()**.

If the following conditions occurs, **_lwp_sigredirect( )** fails and return the corresponding value:

EINVAL The signal *signo* was not pending for the process, or *signo* was not in the notification set.

ESRCH The *target_lwp* cannot be found in the current process.

SEE ALSO

_lwp_create(2), **_lwp_kill**(2), **sigtimedwait**(3R), **siginfo**(5), **signal**(5)

NOTES

This mechanism for delivering signals to multi-threaded processes is subject to change in future versions of Solaris. Any process with explicit knowledge of this mechanism may not be compatible from release to release.

NAME

access – determine accessibility of a file

SYNOPSIS

#include <unistd.h>

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

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

access( ) checks the file pointed to by *path* for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID. This allows a setuid process to verify that the user running it would have had permission to access this file. The bit pattern contained in *amode* is constructed by an OR of the access permissions to be checked (R_OK, W_OK, and X_OK, or the existence test, (F_OK). These constants are defined in <**unistd.h**> as follows:

R_OK Test for read permission.
W_OK Test for write permission.
X_OK Test for execute or search permission.
F_OK Check existence of file

See **intro**(2) for additional information about "File Access Permission".

RETURN VALUES

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

ERRORS

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

EACCES	Search permission is denied on a component of the path prefix.
EACCES	Permission bits of the file mode do not permit the requested access.
EFAULT	*path* points to an illegal address.
EINTR	A signal was caught during the **access( )** function.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	The *path* argument points to an empty string or to the name of a file that does not exist.
ENOLINK	*path* points to a remote machine and the link to that machine is no longer active.

ENOTDIR	A component of the path prefix is not a directory.
EROFS	Write access is requested for a file on a read-only file system.

SEE ALSO **intro**(2), **chmod**(2), **stat**(2)

NAME

acct – enable or disable process accounting

SYNOPSIS

```
#include <unistd.h>
int acct(const char *path);
```

DESCRIPTION

acct( ) enables or disables the system process accounting routine. If the routine is enabled, an accounting record will be written in an accounting file for each process that terminates. The termination of a process can be caused by one of two things: an **exit( )** call or a signal (see **exit**(2) and **signal**(3C)). The effective user ID of the process calling **acct( )** must be super-user.

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

The accounting routine is enabled if *path* is non-zero and no errors occur during the function. It is disabled if *path* is **(char *)NULL** and no errors occur during the function.

RETURN VALUES

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

ERRORS

acct( ) fails if one or more of the following are true:

EACCES	The file named by *path* is not an ordinary file.
EBUSY	An attempt is being made to enable accounting using the same file that is currently being used.
EFAULT	*path* points to an illegal address.
ELOOP	Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	One or more components of the accounting file pathname do not exist.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	The effective user of the calling process is not super-user.
EROFS	The named file resides on a read-only file system.

SEE ALSO

exit(2), **signal**(3C), **acct**(4)

NAME

acl, facl – get or set a file's Access Control List (ACL)

SYNOPSIS

#include <sys/acl.h>

int acl(char *pathp, int cmd, int nentries, aclent_t *aclbufp)

int facl(int fildes, int cmd, int nentries, aclent_t *aclbufp)

DESCRIPTION

acl() and **facl()** get or set the ACL of a file whose name is given by *pathp* or referenced by the open file descriptor *fildes. Nentries* specifies how many ACL entries fit into buffer *aclbufp.* **acl()** is used to manipulate ACLs on file system objects.

The following three values for *cmd* are available.

SETACL

nentries ACL entries, specified in buffer *aclbufp,* are stored in the file's ACL. This command can only be executed by a process that has an effective user ID equal to the owner of the file. All directories in the path name must be searchable.

GETACL

Buffer is filled with the file's ACL entries. Read access to the file is not required, but all directories in the path name must be searchable.

GETACLCNT

The number of entries in the file's ACL is returned. Read access to the file is not required, but all directories in the path name must be searchable.

acl() will fail if one or more of the following is true:

[EACCESS]
The caller does not have access to a component of the pathname.

[EINVAL]
cmd is not GETACL, SETACL, or GETACLCNT.

[EINVAL]
cmd is SETACL and *nentries* is less than three.

[EINVAL]
cmd is SETACL and the ACL specified in *aclbufp* is not valid.

[EIO] A disk i/o error has occurred while storing or retrieving the ACL.

[EPERM]
cmd is SETACL and the effective user ID of the caller does not match the owner of the file.

[ENOENT]
A component of the path does not exist.

[ENOSPC]
cmd is GETACL and *nentries* is less than the number of entries in the file's ACL.

[ENOSPC]

cmd is SETACL and there is insufficient space in the file system to store the ACL.

[ENOTDIR]
A component of the path sepcified by *pathp* is not a directory.

[ENOTDIR]
cmd is SETACL and an attempt is made to set a default ACL on a file type other than a directory.

[ENOSYS]
cmd is SETACL and the file specified by *pathp* resides on a file system that does not support ACLs.

[EROFS]
cmd is SETACL and the file specified by *pathp* resides on a file system that is mounted read-only.

[EFAULT]
pathp or *aclbufp* points to an illegal address.

DIAGNOSTICS

Upon successful completion, if *cmd* is SETACL, a value of 0 is returned. If *cmd* is GETACL or GETACLCNT, the number of ACL entries is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

ENOSYS **acl( )** is not supported by this implementation.

SEE ALSO

getfacl(1), **getfacl**(1), **aclcheck**(3), **aclsort**(3)

NAME

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

SYNOPSIS

```
#include <sys/time.h>
int adjtime(struct timeval *delta, struct timeval *olddelta);
```

DESCRIPTION

adjtime( ) adjusts the system's notion of the current time, as returned by **gettimeofday**(3C), advancing or retarding it by the amount of time specified in the **struct timeval** 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 successful return, the number of seconds and/or 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 VALUES

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

The following error codes may be set in **errno**:

EFAULT	*delta* or *olddelta* points outside the process's allocated address space, or *olddelta* points to a region of the process' allocated address space that is not writable.
EINVAL	*tv_usec* field in *delta* is not within valid range (–1000000 to 1000000).
EPERM	The effective user of the calling process is not super-user.

SEE ALSO

date(1), **gettimeofday**(3C)

NAME

alarm – set a process alarm clock

SYNOPSIS

```
#include <unistd.h>
unsigned alarm(unsigned sec);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

alarm( ) instructs the alarm clock of the calling process to send the signal **SIGALRM** to the calling process after the number of real time seconds specified by *sec* have elapsed (see **signal**(3C)).

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

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

fork(2) sets the alarm clock of a new process to 0. A process created by the exec family of routines inherits the time left on the old process's alarm clock.

Calling **alarm( )** in a multi-threaded process linked with **-lthread** (Solaris threads) and not with **-lpthread** (POSIX threads) currently behaves in the following fashion:

- if the calling thread is a bound thread, the resulting **SIGALRM** is delivered to the bound thread's LWP, i.e. to the calling thread. There is a bug currently that this signal is not maskable via **thr_sigsetmask**(3T) on this bound thread.
- if the calling thread is an unbound thread, the resulting **SIGALRM** is sent to the LWP on which the thread was running when it issued the call to **alarm( )**. This is neither a per-process semantic, nor a per-thread semantic, since the LWP could change threads after the call to **alarm( )** but before the **SIGALRM** delivery, causing some other thread to get it possibly. Hence this is basically a bug.

The above documents current behavior and the bugs are not going to be fixed since the above semantics are going to be discontinued in the next release.

The semantic for Solaris threads will move to the per-process semantic specified by POSIX at this future date. New applications should not rely on the per-thread semantic of **alarm( )**, since this semantic will become obsolete.

In a process linked with **-lpthread** (whether or not it is also linked with **-lthread**), the semantics of **alarm( )** are per-process, i.e. the resulting **SIGALRM** is sent to the process, and not necessarily to the calling thread. This semantic will be supported in the future.

This semantic is obtainable by simply linking with **-lpthread**. One can continue to use Solaris thread interfaces by linking with both **-lpthread** and **-lthread**.

RETURN VALUES

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

SEE ALSO

exec(2), **fork**(2), **pause**(2), **signal**(3C), **thr_sigsetmask**(3T)

NAME

audit – write a record to the audit log

SYNOPSIS

cc [*flag* ...] *file* ... **–lbsm –lsocket –lnsl –lintl** [*library* ...]

```
#include <sys/param.h>
#include <bsm/audit.h>

int audit( caddr_t record, int length);
```

AVAILABILITY

The functionality described in this man page is available only if the Basic Security Module (BSM) has been enabled. See **bsmconv**(1M) for more information.

DESCRIPTION

The **audit** system call is used to write a record to the system audit log. The data pointed to by *record* is written to the log after a minimal consistency check, with the *length* parameter specifying the size of the record in bytes. The data should be a well-formed audit record as described by **audit.log**(4).

The kernel validates the record header token type and length, and sets the time stamp value before writing the record to the audit log. The kernel does not do any preselection for user-level generated events. If the audit policy is set to include sequence or trailer tokens, the kernel will append them to the record.

Only the super-user may successfully execute this call.

RETURN VALUES

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

ERRORS

audit() fails if one or more of the following are true:

EFAULT	*record* points outside the process's allocated address space.
EINVAL	The record header token ID is invalid or the length is either less than the header token size or greater than **MAXAUDITDATA**.
EPERM	The process's effective user ID is not super-user.

SEE ALSO

auditd(1M), **auditon**(2), **auditsvc**(2), **getaudit**(2), **audit.log**(4)

NAME

auditon – manipulate auditing

SYNOPSIS

cc [*flag* ...] *file* ... **–lbsm –lsocket –lnsl –lintl** [*library* ...]

```
#include <sys/param.h>
#include <bsm/audit.h>

int auditon( int cmd, caddr_t data, int length);
```

AVAILABILITY

The functionality described in this man page is available only if the Basic Security Module (BSM) has been enabled. See **bsmconv**(1M) for more information.

DESCRIPTION

The **auditon** system call performs various audit subsystem control operations. The *cmd* argument designates the particular audit control command. The *data* argument is a pointer to command specific data. The *length* argument is the length in bytes of the command specific data.

The following commands are supported:

A_GETCOND
Returns the system audit on / off / disabled condition in the integer long pointed to by *data.*

A_SETCOND
Sets the system's audit on / off condition to the value in the integer long pointed to by *data.* If the current state is disabled, the BSM audit module must be enabled by **bsmconv**(1M) before auditing can be turned on.

A_GETCLASS
Returns the event to class mapping for the designated audit event. The *data* argument points to the **au_evclass_map** structure containing the event number. The preselection class mask is returned in the same structure.

A_SETCLASS
Sets the event class preselection mask for the designated audit event. The *data* argument points to the **au_evclass_map** structure containing the event number and class mask.

A_GETKMASK
Returns the kernel preselection mask in the **au_mask** structure pointed to by *data.*

A_SETKMASK
Sets the kernel preselection mask. The *data* argument points to the **au_mask** structure containing the class mask.

A_GETPINFO
Returns the audit ID, preselection mask, terminal ID and audit session ID of the specified process in the **auditpinfo** structure pointed to by *data.*

A_SETPMASK
Sets the preselection mask of the specified process. The *data* argument points to the **auditpinfo** structure containing the process ID and the preselection mask.

A_SETUMASK
Sets the preselection mask for all processes with the specified audit ID. The *data* argument points to the **auditinfo** structure containing the audit ID and the preselection mask.

A_SETSMASK
Sets the preselection mask for all processes with the specified audit session ID. The *data* argument points to the **auditinfo** structure containing the audit session ID and the preselection mask.

A_GETQCTRL
Returns the kernel audit queue control parameters. These control the high and low water marks of the number of audit records allowed in the audit queue. Another parameter controls the size of the data buffer used by **auditsvc**(2) to write data to the audit trail. There is also a parameter that specifies a delay before data is written to the audit trail. The audit queue parameters are returned in the **au_qctrl** structure pointed to by *data.*

A_SETQCTRL
Sets the kernel audit queue control parameters. The *data* argument points to the **au_qctrl** structure containing the audit queue control parameters.

A_GETCWD
Returns the current working directory as kept by the audit subsystem. This is a path anchored on the real root, rather than on the active root. The *data* argument points to a buffer into which the path is copied. The *length* argument provides the length of the buffer.

A_GETCAR
Returns the current active root as kept by the audit subsystem. This path may be used to anchor an absolute path for a path token generated by an application. The *data* argument points to a buffer into which the path is copied. The *length* argument provides the length of the buffer.

A_GETSTAT
Returns the system audit statistics in the **audit_stat** structure pointed to by *data.*

A_SETSTAT
Resets system audit statistics values.

A_GETPOLICY
Returns the audit policy flags in the integer long pointed to by *data.*

A_SETPOLICY
Sets the audit policy flags to the values in the integer long pointed to by *data.* The following policy flags are recognized:

AUDIT_CNT Do not suspend processes when audit storage is full or inaccessible. The default action is to suspend processes until storage becomes available.

AUDIT_ARGV Include the argument list for the **exec**(2) system call in the audit record. The default action is not to include this information.

AUDIT_ARGE Include the environment variables for the **execv**(2) system call in the audit record. The default action is not to include this information.

AUDIT_SEQ Add a *sequence* token to each audit record. The default action is not to include it.

AUDIT_TRAIL Append a *trailer* token to each audit record. The default action is not to include it.

AUDIT_GROUP Include the supplementary groups list in audit records. The default action is not to include it.

AUDIT_PATH Include secondary paths in audit records. Examples of secondary paths are dynamically loaded shared library modules and the command shell path for executable scripts.

Only the super-user may successfully execute this call.

RETURN VALUES

auditon( ) returns:

0 on success.

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

ERRORS

EFAULT The copy of data to/from the kernel failed.

EINVAL One of the system call arguments was illegal.

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

SEE ALSO

auditd(1M), **bsmconv**(1M), **audit**(2), **auditsvc**(2), **audit.log**(4)

NAME

auditsvc – write audit log to specified file descriptor

SYNOPSIS

cc [*flag* ...] *file* ... **–lbsm –lsocket –lnsl –lintl** [*library* ...]

```
#include <sys/param.h>
#include <bsm/audit.h>

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

AVAILABILITY

The functionality described in this man page is available only if the Basic Security Module (BSM) has been enabled. See **bsmconv**(1M) for more information.

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 the number of free blocks that must be available in the audit file system, and causes **auditsvc( )** to return when the 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 an 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.
ENOSPC	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**(3B)). **signal**(3C), 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 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 **auditd**(1M), **audit**(2), **auditon**(2), **sigvec**(3B), **audit.log**(4)

NAME

brk, sbrk – change the amount of space allocated for the calling process's data segment

SYNOPSIS

```
#include <unistd.h>
int brk(void *endds);
void *sbrk(int incr);
```

DESCRIPTION

brk( ) and **sbrk( )** are used to change dynamically the amount of space allocated for the calling process's data segment (see **exec**(2)). The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. Newly allocated space is set to zero. If, however, the same memory space is reallocated to the same process its contents are undefined.

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) function 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**(3C) for the definition of **etext( )**.)

brk( ) sets the break value to *endds* and changes the allocated space accordingly.

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

RETURN VALUES

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

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.
ENOMEM	The maximum possible size of a data segment (compiled into the system) would be exceeded.
ENOMEM	Insufficient space exists in the swap area to support the expansion.
ENOMEM	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)).
EAGAIN	Total amount of system memory available for private pages is temporarily insufficient. This may occur even though the space requested was less than the maximum data segment size (see **ulimit**(2)).

SEE ALSO **exec**(2), **getrlimit**(2), **mmap**(2), **shmop**(2), **ulimit**(2), **end**(3C), **malloc**(3C)

WARNINGS Programs combining the **brk( )** and **sbrk( )** functions 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.

NOTES The value of *incr* may be adjusted by the system before setting the new break value. Upon successful completion, the implementation guarantees a minimum of *incr* bytes will be added to the data segment if *incr* is a positive value. If *incr* is a negative value, a maximum of *incr* bytes will be removed from the data segment. This adjustment may not be necessary for all machine architectures.

The value of the arguments to both **brk( )** and **sbrk( )** are rounded up for alignment with eight-byte boundries.

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, fchdir – change working directory

SYNOPSIS

```
#include <unistd.h>
int chdir(const char *path);
int fchdir(int fildes);
```

MT-LEVEL

chdir() is Async-Signal-Safe

DESCRIPTION

chdir() and **fchdir()** cause a directory pointed to by *path* or *fildes* to become the current working directory. The starting point for path searches for path names not beginning with **/**. *path* points to the path name of a directory. The *fildes* argument to **fchdir()** is an open file descriptor of a directory.

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

RETURN VALUES

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

ERRORS

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

EACCES	Search permission is denied for any component of the path name.
EFAULT	*path* points to an illegal address.
EINTR	A signal was caught during the execution of the **chdir()** function.
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**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	Either a component of the path prefix or the directory named by *path* does not exist or is a null pathname.
ENOLINK	*path* points to a remote machine and the link to that machine is no longer active.
ENOTDIR	A component of the path name is not a directory.
EMULTIHOP	Components of *path* require hopping to multiple remote machines and file system type does not allow it.

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

EACCES	Search permission is denied for *fildes*.
EBADF	*fildes* is not an open file descriptor.
EINTR	A signal was caught during the execution of the **fchdir( )** function.
EIO	An I/O error occurred while reading from or writing to the file system.
ENOLINK	*fildes* points to a remote machine and the link to that machine is no longer active.
ENOTDIR	The open file descriptor *fildes* does not refer to a directory.

SEE ALSO **chroot**(2)

NAME

chmod, fchmod – change access permission mode of file

SYNOPSIS

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

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

int fchmod(int fildes, mode_t mode);
```

MT-LEVEL

chmod( ) is Async-Signal-Safe

DESCRIPTION

chmod( ) and **fchmod( )** set the access permission portion of the mode of the file whose name is given by *path* or referenced by the open file descriptor *fildes* to the bit pattern contained in *mode*. Access permission bits are interpreted as follows:

S_ISUID	04000	Set user ID on execution.
S_ISGID	020#0	Set group ID on execution if # is **7**, **5**, **3**, or **1**. Enable mandatory file / record locking if # is **6**, **4**, **2**, or **0**.
S_ISVTX	01000	Save text image after execution.
S_IRWXU	00700	Read, write, execute by owner.
S_IRUSR	00400	Read by owner.
S_IWUSR	00200	Write by owner.
S_IXUSR	00100	Execute (search if a directory) by owner.
S_IRWXG	00070	Read, write, execute by group.
S_IRGRP	00040	Read by group.
S_IWGRP	00020	Write by group.
S_IXGRP	00010	Execute by group.
S_IRWXO	00007	Read, write, execute (search) by others.
S_IROTH	00004	Read by others.
S_IWOTH	00002	Write by others.
S_IXOTH	00001	Execute by others.

Modes are constructed by **OR**'ing the access permission bits.

The effective user ID of the process must match the owner of the file or the process must have the appropriate privilege to change the mode of a file.

If the process is not a privileged process and the file is not a directory, mode bit 01000 (save text image on execution) is cleared.

If neither the process is privileged, nor the file's group is a member of the process's supplementary group list, and the effective group ID of the process does not match the group ID of the file, mode bit 02000 (set group ID on execution) is cleared.

If a directory is writable and has S_ISVTX (the sticky bit) set, files within that directory can be removed or renamed only if one or more of the following is true (see **unlink**(2) and **rename**(2)):

- the user owns the file
- the user owns the directory
- the file is writable by the user
- the user is a privileged user

If a directory has the set group ID bit set, a given file created within that directory will have the same group ID as the directory, if that group ID is part of the group ID set of the process that created the file. Otherwise, the newly created file's group ID will be set to the effective group ID of the creating process.

If the mode bit 02000 (set group ID on execution) is set and the mode bit 00010 (execute or search by group) is not set, mandatory file/record locking will exist on a regular file. This may affect future calls to **open**(2), **creat**(2), **read**(2), and **write**(2) on this file.

Upon successful completion, **chmod()** and **fchmod()** mark for update the **st_ctime** field of the file.

RETURN VALUES

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

ERRORS

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

EACCES	Search permission is denied on a component of the path prefix of *path*.
EFAULT	*path* points to an illegal address.
EINTR	A signal was caught during execution of the function.
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*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines and file system type does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	Either a component of the path prefix, or the file referred to by *path* does not exist or is a null pathname.
ENOLINK	*fildes* points to a remote machine and the link to that machine is no longer active.
ENOTDIR	A component of the prefix of *path* is not a directory.

EPERM	The effective user ID does not match the owner of the file and is not super-user.
EROFS	The file referred to by *path* resides on a read-only file system.

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

EBADF	*fildes* is not an open file descriptor
EIO	An I/O error occurred while reading from or writing to the file system.
EINTR	A signal was caught during execution of the **fchmod()** function.
ENOLINK	*path* points to a remote machine and the link to that machine is no longer active.
EPERM	The effective user ID does not match the owner of the file and the process does not have appropriate privilege.
EROFS	The file referred to by *fildes* resides on a read-only file system.

SEE ALSO

chmod(1), **chown**(2), **creat**(2), **fcntl**(2), **mknod**(2), **open**(2), **read**(2), **rename**(2), **stat**(2), **write**(2), **mkfifo**(3C), **stat**(5)

System Interfaces Guide

NAME

chown, lchown, fchown – change owner and group of a file

SYNOPSIS

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

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

int lchown(const char *path, uid_t owner, gid_t group);

int fchown(int fildes, uid_t owner, gid_t group);
```

MT-LEVEL

chown() is Async-Signal-Safe

DESCRIPTION

chown() sets the owner ID and group ID of the file specified by *path* or referenced by the open file descriptor *fildes* to *owner* and *group* respectively. If *owner* or *group* is specified as –1, **chown()** does not change the corresponding ID of the file.

The function **lchown()** sets the owner ID and group ID of the named file just as **chown()** does, except in the case where the named file is a symbolic link. In this case, **lchown()** changes the ownership of the symbolic link file itself, while **chown()** changes the ownership of the file or directory to which the symbolic link refers.

If **chown()**, **lchown()**, or **fchown()** is invoked by a process other than super-user, the set-user-ID and set-group-ID bits of the file mode, **S_ISUID** and **S_ISGID** respectively, are cleared (see **chmod**(2)).

The operating system has a configuration option, {**_POSIX_CHOWN_RESTRICTED**}, to restrict ownership changes for the **chown()**, **lchown()**, and **fchown()** functions. When {**_POSIX_CHOWN_RESTRICTED**} is not in effect, the effective user ID of the process must match the owner of the file or the process must be the super-user to change the ownership of a file. When {**_POSIX_CHOWN_RESTRICTED**} is in effect, the **chown()**, **lchown()**, and **fchown()** functions, for users other than super-user, prevent the owner of the file from changing the owner ID of the file and restrict the change of the group of the file to the list of supplementary group IDs.

Upon successful completion, **chown()**, **fchown()** and **lchown()** mark for update the **st_ctime** field of the file.

RETURN VALUES

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

ERRORS

chown() and **lchown()** fail and the owner and group of the named file remain unchanged if one or more of the following are true:

EACCES	Search permission is denied on a component of the path prefix of *path*.
EFAULT	*path* points to an illegal address.
EINTR	A signal was caught during the **chown()** or **lchown()** functions.
EINVAL	*group* or *owner* is out of range.

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*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines and file system type does not allow it. Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOLINK	*path* points to a remote machine and the link to that machine is no longer active.
ENOENT	Either a component of the path prefix or the file referred to by *path* does not exist or is a null pathname.
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 or the process is not the super-user and {**_POSIX_CHOWN_RESTRICTED**} indicates that such privilege is required.
EROFS	The named file resides on a read-only file system.

fchown() fails and the owner and group of the named file remain unchanged if one or more of the following are true:

EBADF	*fildes* is not an open file descriptor.
EIO	An I/O error occurred while reading from or writing to the file system.
EINTR	A signal was caught during execution of the function.
ENOLINK	*fildes* points to a remote machine and the link to that machine is no longer active.
EINVAL	*group* or *owner* is out of range.
EPERM	The effective user ID does not match the owner of the file or the process is not the super-user and {**_POSIX_CHOWN_RESTRICTED**} indicates that such privilege is required.
EROFS	The named file referred to by *fildes* resides on a read-only file system.

SEE ALSO **chgrp**(1), **chown**(1), **chmod**(2)

NAME

chroot, fchroot – change root directory

SYNOPSIS

#include <unistd.h>

int chroot(const char **path*);

int fchroot(int *fildes*);

DESCRIPTION

chroot() and **fchroot()** cause a directory to become the root directory, the starting point for path searches for path names beginning with **/**. The user's working directory is unaffected by the **chroot()** and **fchroot()** functions.

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

The effective user ID of the process must be super-user to change the root directory. **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 *fildes* be valid in all respects.

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.

RETURN VALUES

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

ERRORS

chroot() will fail and the root directory will remain 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	*path* points to an illegal address.
EINVAL	**fchroot()** attempted to change to a directory which is not the system root and external circumstances do not allow this.
EINTR	A signal was caught during the **chroot()** function.
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*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines and file system type does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the

	length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	The named directory does not exist or is a null pathname.
ENOLINK	*path* points to a remote machine and the link to that machine is no longer active.
ENOTDIR	Any component of the path name is not a directory.
EPERM	The effective user of the calling process is not super-user.

SEE ALSO **chroot**(1M), **chdir**(2)

WARNINGS The only use of **fchroot()** that is appropriate is to change back to the system root.

NAME

close – close a file descriptor

SYNOPSIS

```
#include <unistd.h>
int close(int fildes);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

close( ) closes the file descriptor indicated by *fildes*. All outstanding record locks owned by the process (on the file indicated by *fildes*) are removed. *fildes* is an open file descriptor.

When all file descriptors associated with the open file description have been closed, the open file description is freed.

If the link count of the file is zero, when all file descriptors associated with the file have been closed, the space occupied by the file is freed and the file is no longer accessible.

If a STREAMS-based (see **intro**(2)) *fildes* is closed, and the calling process had previously registered to receive a **SIGPOLL** signal (see **signal**(3C)) for events associated with that stream (see **I_SETSIG** in **streamio**(7I)), the calling process will be unregistered for events associated with the stream. The last **close( )** for a stream causes the stream associated with *fildes* to be dismantled. If **O_NDELAY** and **O_NONBLOCK** are clear and there have been no signals posted for the stream, and if there are data on the module's write queue, **close( )** waits up to 15 seconds (for each module and driver) for any output to drain before dismantling the stream. The time delay can be changed using an **I_SETCLTIME ioctl** request (see **streamio**(7I)). If **O_NDELAY** or **O_NONBLOCK** is set, or if there are any pending signals, **close( )** does not wait for output to drain, and dismantles the stream immediately.

If *fildes* is associated with one end of a pipe, the last **close( )** causes a hangup to occur on the other end of the pipe. In addition, if the other end of the pipe has been named (see **fattach**(3C)), the last **close( )** forces the named end to be detached (see **fdetach**(3C)). If the named end has no open processes associated with it and becomes detached, the stream associated with that end is also dismantled.

RETURN VALUES

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

ERRORS

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

EBADF	*fildes* is not a valid open file descriptor.
EINTR	A signal was caught during the **close( )** function.
ENOLINK	*fildes* is on a remote machine and the link to that machine is no longer active.
EIO	Data was not written out properly.

SEE ALSO

intro(2), **creat**(2), **dup**(2), **exec**(2), **fcntl**(2), **open**(2), **pipe**(2), **fattach**(3C), **fdetach**(3C), **signal**(3C), **signal**(5), **streamio**(7I)

See ‘‘File Description’’ in **intro**(2).

NAME

creat – create a new file or rewrite an existing one

SYNOPSIS

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

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

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

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

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

If the file does not exist the file's owner ID is set to the effective user ID of the process. The group ID of the file is set to the effective group ID of the process, or if the **S_ISGID** bit is set in the parent directory then the group ID of the file is inherited from the parent directory. The access permission bits of the file mode are set to the value of *mode* modified as follows:

- If the group ID of the new file does not match the effective group ID or one of the supplementary group IDs, the **S_ISGID** bit is cleared.
- All bits set in the process's file mode creation mask are cleared (see **umask**(2)).
- The "save text image after execution bit" of the mode is cleared (see **chmod**(2) for the values of mode).

Upon successful completion, a write-only file descriptor is returned and the file is open for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across **exec** functions (see **fcntl**(2)). A new file may be created with a mode that forbids writing.

The call **creat(***path***,** *mode***)** is equivalent to:

open(*path***, O_WRONLY | O_CREAT | O_TRUNC,** *mode***)**

RETURN VALUES

Upon successful completion a non-negative integer, namely the lowest numbered unused file descriptor, is returned. Otherwise, a value of –1 is returned, no files are created or modified, and **errno** is set to indicate the error.

ERRORS

creat( ) fails if one or more of the following are true:

EACCES	Search permission is denied on a component of the path prefix.
EACCES	The file does not exist and the directory in which the file is to be created does not permit writing.
EACCES	The file exists and write permission is denied.
EAGAIN	The file exists, mandatory file / record locking is set, and there are outstanding record locks on the file (see **chmod**(2)).
EFAULT	*path* points to an illegal address.

EINTR	A signal was caught during the **creat( )** function.
EISDIR	The named file is an existing directory.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMFILE	The process has too many open files (see **getrlimit**(2)).
EMULTIHOP	Components of *path* require hopping to multiple remote machines.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENFILE	The system file table is full.
ENOENT	A component of the path prefix does not exist.
ENOENT	The path name is null.
ENOLINK	*path* points to a remote machine and the link to that machine is no longer active.
ENOSPC	The file system is out of inodes.
ENOTDIR	A component of the path prefix is not a directory.
EROFS	The named file resides or would reside on a read-only file system.

SEE ALSO **chmod**(2), **close**(2), **dup**(2), **fcntl**(2), **getrlimit**(2), **lseek**(2), **open**(2), **read**(2), **umask**(2), **write**(2), **stat**(5)

NAME

door, door_call, door_create, door_info, door_return, door_revoke – Solaris 2.5 internal implementation detail

SYNOPSIS

int door_call( );

int door_create( );

int door_info( );

int door_return( );

int door_revoke( );

DESCRIPTION

This family of system calls provide a new flavor of interprocess communication between client and server processes. The doors mechanism is not yet available for public consumption because the interface is still evolving and will undergo a major rework in a future release of Solaris. In Solaris 2.5, doors are used as part of the implementation of the name service cache daemon, **nscd**(1M).

SEE ALSO

truss(1), **nscd**(1M)

WARNING

Please do not attempt to reverse-engineer the interface and program to it. If you do, your program will almost certainly fail to run on future versions of Solaris, and may even be broken by a patch. This document does not constitute an API. Doors may not exist or may have a completely different set of semantics in a future release.

NOTES

This manual page is here solely for the benefit of anyone who noticed **door_call**() in **truss**(1) output and thought, "Gee, I wonder what that does..."

Processes that appear to be "stuck" waiting in **door_call**(2) are probably waiting for **nscd**(1M) to reply to a name service lookup.

NAME

dup – duplicate an open file descriptor

SYNOPSIS

```
#include <unistd.h>
int dup(int fildes);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

dup() returns a new file descriptor having the following in common with the original open file descriptor *fildes*:

Same open file (or pipe).

Same file pointer (that is, both file descriptors share one file pointer).

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

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

The file descriptor returned is the lowest one available.

The **dup(***fildes***)** is equivalent to

fcntl(*fildes***, F_DUPFD, 0)**

RETURN VALUES

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

ERRORS

dup() will fail if one or more of the following are true:

EBADF	*fildes* is not a valid open file descriptor.
EINTR	A signal was caught during the **dup()** function.
EMFILE	The process has too many open files (see **getrlimit**(2)).
ENOLINK	*fildes* is on a remote machine and the link to that machine is no longer active.

SEE ALSO

close(2), **creat**(2), **exec**(2), **fcntl**(2), **getrlimit**(2), **open**(2), **pipe**(2), **dup2**(3C), **lockf**(3C)

NAME

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

SYNOPSIS

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ..., const char *argn, char * /*NULL*/);
int execv(const char *path, char *const argv[]);
int execle (const char *path,char *const arg0[], ..., const char *argn,
        char * /*NULL*/, char *const envp[]);
int execve (const char *path, char *const argv[], char *const envp[]);
int execlp (const char *file, const char *arg0, ..., const char *argn, char * /*NULL*/);
int execvp (const char *file, char *const argv[]);
```

MT-LEVEL

execle( ) and **execve( )** are Async-Signal-Safe

DESCRIPTION

exec( ) in all its forms overlays a new process image on an old process. The new process image is constructed from an ordinary, executable file. This file is either an executable object file, or a file of data for an interpreter. There can be no return from a successful **exec( )** because the calling process image is overlaid by the new process image.

An interpreter file begins with a line of the form

#! *pathname* [*arg*]

where *pathname* is the path of the interpreter, and *arg* is an optional argument. When an interpreter file is exec'd, the system execs the specified interpreter. The pathname specified in the interpreter file is passed as *arg0* to the interpreter. If *arg* was specified in the interpreter file, it is passed as *arg1* to the interpreter. The remaining arguments to the interpreter are *arg0* through *argn* of the originally exec'd file.

When a C program is executed, it is called as follows:

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

where *argc* is the argument count, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. As indicated, *argc* is at least one, and the first member of the array points to a string containing the name of the file.

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

file points to the new process file. If *file* does not contain a slash character, the path prefix for this file is obtained by a search of the directories passed in the **PATH** environment variable (see **environ**(5)). The environment is supplied typically by the shell. If the new process file is not an executable object file, **execlp( )** and **execvp( )** use the contents of that file as standard input to the shell.

Solaris **exec( )** uses **/usr/bin/sh** (see **sh**(1)).

XPG4 **exec( )** uses the XPG4-compliant shell **/usr/bin/ksh** (see **ksh**(1)).

The arguments *arg0*, ..., *argn* point to null-terminated character strings. These strings constitute the argument list available to the new process image. Conventionally at least *arg0* should be present. It will become the name of the process, as displayed by the **ps** command. *arg0* points to a string that is the same as *path* (or the last component of *path*). The list of argument strings is terminated by a **(char *)0** argument.

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

envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. *envp* is terminated by a null pointer. For **execl(), execv(), execvp(),** and **execlp(),** the C run-time start-off routine places a pointer to the environment of the calling process in the global object **extern char** ***environ**, and it is used to pass the environment of the calling process to the new process.

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

Signals that are being caught by the calling process are set to the default disposition in the new process image (see **signal**(3C)). Otherwise, the new process image inherits the signal dispositions of the calling process.

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

If the effective user-ID is **root** or super-user, the set-user-ID and set-group-ID bits will be honored when the process is being controlled by **ptrace**.

The shared memory segments attached to the calling process will not be attached to the new process (see **shmop**(2)). Memory mappings in the calling process are unmapped before the new process begins execution (see **mmap**(2)).

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

Timers created by **timer_create**(3R) are deleted before the new process begins execution.

Any outstanding asynchronous I/O operations may be cancelled.

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

nice value (see **nice**(2))
scheduler class and priority (see **priocntl**(2))
process ID
parent process ID
process group ID
supplementary group IDs

semadj values (see **semop**(2))
session ID (see **exit**(2) and **signal**(3C))
trace flag (see **ptrace**(2) request 0)
time left until an alarm (see **alarm**(2))
current working directory
root directory
file mode creation mask (see **umask**(2))
resource limits (see **getrlimit**(2))
utime, **stime**, **cutime**, and **cstime** (see **times**(2))
file-locks (see **fcntl**(2) and **lockf**(3C))
controlling terminal
process signal mask (see **sigprocmask**(2))
pending signals (see **sigpending**(2))

Upon successful completion, **exec( )** marks for update the **st_atime** field of the file, unless the file is on a read-only file system. Should the **exec( )** succeed, the process image file is considered to have been **open**( ) -ed. The corresponding **close**( ) is considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to **exec( ).**

RETURN VALUES

If **exec( )** returns to the calling process, an error has occurred; the return value is –1 and **errno** is set to indicate the error.

ERRORS

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

E2BIG	The number of bytes in the new process's argument list is greater than the system-imposed limit of **ARG_MAX** bytes. The argument list limit is sum of the size of the argument list plus the size of the environment's exported shell variables.
EACCES	Search permission is denied for a directory listed in the new process file's path prefix.
EACCES	The new process file is not an ordinary file.
EACCES	The new process file mode denies execute permission.
EAGAIN	Total amount of system memory available when reading using raw I/O is temporarily insufficient.
EFAULT	An argument points to an illegal address.
EINTR	A signal was caught during the **exec( )** function.
ELOOP	Too many symbolic links were encountered in translating *path* or *file.*
EMULTIHOP	Components of *path* require hopping to multiple remote machines and the file system type does not allow it.
ENAMETOOLONG	The length of the *file* or *path* argument exceeds {**PATH_MAX**}, or the length of a *file* or *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.

ENOENT	One or more components of the new process path name of the file do not exist or is a null pathname.
ENOEXEC	The **exec( )** is not an **execlp( )** or **execvp( ),** and the new process file has the appropriate access permission but an invalid magic number in its header.
ENOLINK	*path* points to a remote machine and the link to that machine is no longer active.
ENOMEM	The new process requires more memory than is allowed by the limit imposed by **getrlimit( )**, see **brk**(2). MAXMEM.
ENOTDIR	A component of the new process path of the file prefix is not a directory.

SEE ALSO

ksh(1), **ps**(1), **sh**(1), **alarm**(2), **brk**(2), **chmod**(2), **exit**(2), **fcntl**(2), **fork**(2), **getrlimit**(2), **mmap**(2), **nice**(2), **priocntl**(2), **profil**(2), **ptrace**(2), **semop**(2), **shmop**(2), **signal**(3C), **sigpending**(2), **sigprocmask**(2), **times**(2), **umask**(2), **lockf**(3C), **timer_create**(3R), **system**(3S), **a.out**(4), **environ**(5), **xpg4**(5)

WARNINGS

If a program is **setuid** to a user ID other than the super-user, and the program 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, _exit – terminate process

SYNOPSIS

#include <unistd.h>

void _exit(int *status***);**

MT-LEVEL

_exit( ) is Async-Signal-Safe

DESCRIPTION

_exit( ) terminates the calling process with the following consequences:

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

A **SIGCHLD** signal is sent to the calling process's parent process.

If the parent process of the calling process has "not" specified the **SA_NOCLDWAIT** flag (see **sigaction**(2)),

the calling process is transformed into a "zombie process." A zombie process is a process that only occupies a slot in the process table. It has no other space allocated either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information (see **<sys/proc.h>**) to be used by the **times** function.

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

If the parent process of the calling process is not executing a **wait**(2) or **waitpid**(2), *status* is saved for return to the parent process whenever the parent process executes an appropriate subsequent **wait**(2) or

waitpid(2).

Otherwise, if the parent process of the calling process has specified the **SA_NOCLDWAIT** flag,

status is unavailable to the parent.

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

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. Memory mappings created in the process are unmapped.

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

If the process has process, text or data locks, an *unlock* is performed (see **plock**(3C) and **memcntl**(2)).

All open named semaphores in the process are closed as if by appropriate calls to **sem_close**(3R). All open message queues in the process are closed as if by appropriate calls to **mq_close**(3R). Any outstanding asynchronous I/O operations

may be cancelled.

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

If the process is a controlling process, **SIGHUP** is sent to the foreground process group of its controlling terminal and its controlling terminal is deallocated.

If the calling process has any stopped children whose process group will be orphaned when the calling process exits, or if the calling process is a member of a process group that will be orphaned when the calling process exits, that process group will be sent **SIGHUP** and **SIGCONT** signals.

SEE ALSO **acct**(2), **intro**(2), **memcntl**(2), **semop**(2), **sigaction**(2), **times**(2), **wait**(2), **waitpid**(2), **exit**(3C), **plock**(3C), **mq_close**(3R), **sem_close**(3R), **signal**(5)

NAME

fcntl – file control

SYNOPSIS

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

int fcntl(int fildes, int cmd, /* arg */ ...);
```

MT-LEVEL

Async_Signal_Safe

DESCRIPTION

fcntl( ) provides for control over open files. *fildes* is an open file descriptor (see **intro**(2)).

fcntl( ) may take a third argument, *arg*, whose data type, value and use depend upon the value of *cmd*. *cmd* specifies the operation to be performed by **fcntl( )** and may be one of the following:

F_DUPFD Return a new file descriptor with the following characteristics:

Lowest numbered available file descriptor greater than or equal to the integer value given as the third argument.

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

Same file pointer as the original file (that is, both file descriptors share one file pointer).

Same access mode (read, write, or read / write) as the original file.

Shares any locks associated with the original file descriptor.

Same file status flags (that is, both file descriptors share the same file status flags) as the original file.

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

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

F_SETFD Set the close-on-exec flag associated with *fildes* to the low-order bit of the integer value given as the third argument (**0** or **1** as above).

F_GETFL Get *fildes* status flags.

F_SETFL Set *fildes* status flags to the integer value given as the third argument. Only certain flags can be set (see **fcntl**(5)).

F_FREESP Free storage space associated with a section of the ordinary file *fildes*. The section is specified by a variable of data type **struct flock** pointed to by the third argument *arg*. The data type **struct flock** is defined in the **<fcntl.h>** header (see **fcntl**(5)) and contains the following members: **l_whence** is 0, 1, or 2 to indicate that the relative offset **l_start** will be measured from the start of the file, the current position, or the end of the file, respectively. **l_start** is the offset from the position specified in **l_whence**. **l_len** is the size of the section.

An **l_len** of 0 frees up to the end of the file; in this case, the end of file (that is, file size) is set to the beginning of the section freed. Any data previously written into this section is no longer accessible.

Note that all filesystems might not support all possible variations of F_FREESP arguments. In particular, many filesystems only allow space to be freed at the end of a file.

The following values for *cmd* are used for record-locking. Locks may be placed on an entire file or on segments of a file.

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

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

F_GETLK If the lock request described by the **flock** structure that *arg* points to could be created, then the structure is passed back unchanged except that the lock type is set to **F_UNLCK** and the **l_whence** field will be set to **SEEK_SET**.

If a lock is found that would prevent this lock from being created, then the structure is overwritten with a description of the first lock that is preventing such a lock from being created. The structure also contains the process ID and the system ID of the process holding the lock.

This command never creates a lock; it tests whether a particular lock could be created.

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

A write lock prevents any process from read locking or write locking the protected area. Only one write lock and no read locks may exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access.

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

```
typedef struct flock {
        short     l_type;
        short     l_whence;
        off_t     l_start;
        off_t     l_len;      /* len == 0 means until end of file */
        long      l_sysid;
        pid_t     l_pid;
```

```
        long    pad[4];     /* reserve area */
} flock_t;
```

The **flock** structure describes the type (**l_type**), starting offset (**l_whence**), relative offset (**l_start**), size (**l_len**), process ID (**l_pid**), and system ID (**l_sysid**) of the segment of the file to be affected. The process ID and system ID fields are used only with the **F_GETLK** *cmd* to return the values for a blocking lock. Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end of file by setting **l_len** to **0**. If such a lock also has **l_whence** and **l_start** set to **0**, the whole file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments at either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process in a **fork**(2) function.

When mandatory file and record locking is active on a file (see **chmod**(2)), **creat**(2), **open**(2), **read**(2) and **write**(2) functions issued on the file will be affected by the record locks in effect. When mandatory file and record locking is active on a file, it cannot be memory mapped.

RETURN VALUES

On success, **fcntl( )** returns a value that depends on *cmd*:

F_DUPFD	A new file descriptor.
F_GETFD	Value of flag (only the low-order bit is defined). The return value will not be negative.
F_SETFD	Value other than –1.
F_FREESP	Value of 0.
F_GETFL	Value of file status flags. The return value will not be negative.
F_SETFL	Value other than –1.
F_GETLK	Value other than –1.
F_SETLK	Value other than –1.
F_SETLKW	Value other than –1.

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

ERRORS

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

EAGAIN *cmd* is **F_SETLK**, the type of lock (**l_type**) is a read lock (**F_RDLCK**) and the segment of a file to be locked is already write locked by another process, or the type is a write lock (**F_WRLCK**) and the segment of a file to be locked is already read or write locked by another process.
Note that in the past this function was returned as **EACCES**.

EAGAIN *cmd* is **F_FREESP**, the file exists, mandatory file/record locking is set, and there are outstanding record locks on the file.

EAGAIN	*cmd* is **F_SETLK** or **F_SETLKW**, mandatory file/record locking is set, and the file is currently being mapped to virtual memory using **mmap**(2).
EBADF	*fildes* is not a valid open file descriptor.
EBADF	*cmd* is **F_SETLK** or **F_SETLKW**, the type of lock (**l_type**) is a read lock (**F_RDLCK**), and *fildes* is not a valid file descriptor open for reading.
EBADF	*cmd* is **F_SETLK** or **F_SETLKW**, the type of lock (**l_type**) is a write lock (**F_WRLCK**), and *fildes* is not a valid file descriptor open for writing.
EBADF	*cmd* is **F_FREESP**, and *fildes* is not a valid file descriptor open for writing.
EDEADLK	*cmd* is **F_SETLKW**, the lock is blocked by some lock from another process, and if **fcntl( )** blocked the calling process waiting for that lock to become free, a deadlock would occur.
EDEADLK	*cmd* is **F_FREESP**, mandatory record locking is enabled, **O_NDELAY** and **O_NONBLOCK** are clear and a deadlock condition was detected.
EFAULT	*cmd* is **F_FREESP** and the third argument *arg* points to an illegal address.
EFAULT	*cmd* is **F_GETLK**, **F_SETLK** or **F_SETLKW** and the third argument points to an illegal address.
EINTR	A signal was caught during execution of the **fcntl( )** function.
EINVAL	*cmd* is **F_DUPFD** and the third argument is either negative, or greater than or equal to the configured value for the maximum number of open file descriptors allowed each user.
EINVAL	*cmd* is not a valid value.
EINVAL	*cmd* is **F_GETLK**, **F_SETLK**, or **F_SETLKW** and the third argument or the data it points to is not valid, or *fildes* refers to a file that does not support locking.
EIO	An I/O error occurred while reading from or writing to the file system.
EMFILE	*cmd* is **F_DUPFD** and the number of file descriptors currently open in the calling process is the configured value for the maximum number of open file descriptors allowed each user.
ENOLCK	*cmd* is **F_SETLK** or **F_SETLKW**, the type of lock is a read or write lock, and there are no more record locks available (too many file segments locked) because the system maximum has been exceeded.
ENOLINK	*fildes* is on a remote machine and the link to that machine is no longer active.
ENOLINK	*cmd* is **F_FREESP**, the file is on a remote machine, and the link to that machine is no longer active.
EOVERFLOW	*cmd* is **F_GETLK** and the process ID of the process holding the requested lock is too large to be stored in the *l_pid* field.

SEE ALSO

lockd(1M), **chmod**(2), **close**(2), **creat**(2), **dup**(2), **exec**(2), **fork**(2), **open**(2), **pipe**(2), **read**(2), **write**(2), **fcntl**(5)

System Interfaces Guide

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

In the past, the variable **errno** was set to **EACCES** rather than **EAGAIN** when a section of a file is already locked by another process. Therefore, portable application programs should expect and test for either value.

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

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

NAME

fork, fork1 – create a new process

SYNOPSIS

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

pid_t fork(void);

pid_t fork1(void);
```

MT-LEVEL

fork( ) is Async-Signal-Safe

DESCRIPTION

fork( ) and **fork1( )** cause creation of a new process. The new process (child process) is an exact copy of the calling process (parent process). The child process inherits the following attributes from the parent process:

- real user ID, real group ID, effective user ID, effective group ID
- environment
- open file descriptors
- close-on-exec flags (see **exec**(2))
- signal handling settings (that is, **SIG_DFL , SIG_IGN , SIG_HOLD ,** function address)
- supplementary group IDs
- set-user-ID mode bit
- set-group-ID mode bit
- profiling on / off status
- nice value (see **nice**(2))
- scheduler class (see **priocntl**(2))
- all attached shared memory segments (see **shmop**(2))
- process group ID -- memory mappings (see **mmap**(2))
- session ID (see **exit**(2))
- current working directory
- root directory
- file mode creation mask (see **umask**(2))
- resource limits (see **getrlimit**(2))
- controlling terminal
- saved user ID and group ID

Scheduling priority and any per-process scheduling parameters that are specific to a given scheduling class may or may not be inherited according to the policy of that particular class (see **priocntl**(2)). The child process differs from the parent process in the following ways:

- The child process has a unique process ID which does not match any active process group ID.
- The child process has a different parent process ID (that is, the process ID of the parent process).
- The child process has its own copy of the parent's file descriptors and directory streams. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

- Each shared memory segment remains attached and the value of **shm_nattach** is incremented by 1.
- All **semadj** values are cleared (see **semop**(2)).
- Process locks, text locks, data locks, and other memory locks are not inherited by the child (see **plock**(3C) and **memcntl**(2)).
- The child process's **tms** structure is cleared: **tms_utime**, **stime**, **cutime**, and **cstime** are set to 0 (see **times**(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 set of signals pending for the child process is initialized to the empty set.
- Timers created by **timer_create**(3R) are not inherited by the child process.
- No asynchronous input or asynchronous output operations are inherited by the child.

Record locks set by the parent process are not inherited by the child process (see **fcntl**(2)).

MT fork() Solaris Threads

The following are the **fork( )** semantics in programs that use the Solaris threads API rather than the POSIX threads API (programs linked with **–lthread** but not **–lpthread**):

fork( ) duplicates all the threads (see **thr_create**(3T)) and LWPs in the parent process in the child process. **fork1( )** duplicates only the calling thread (LWP) in the child process.

POSIX Threads

The following are the **fork( )** semantics in programs that use the POSIX threads API rather than the Solaris threads API (programs linked with **–lpthread** but not **–lthread**):

The call to **fork( )** is like a call to **fork1( )**, which replicates only the calling thread. There is no call that forks a child with all threads and LWPs duplicated in the child.

Note that if a program is linked with both libraries (**–lthread** and **–lpthread**), the POSIX semantic of **fork( )** prevails.

Fork-safety

If **fork1( )** is called in a Solaris thread program or **fork( )** is called in a POSIX thread program, and the child does more than just call **exec( )**, there is a possibility of deadlocking in the child. To ensure that the application is safe with respect to this deadlock, it should use **pthread_atfork**(3T). Should there be any outstanding mutexes throughout the process, the application should call **pthread_atfork**(3T), to wait for and acquire those mutexes, prior to calling **fork( )**. (See **Intro**(3), "MT-Level of Libraries")

RETURN VALUES

Upon successful completion, **fork( )** and **fork1( )** returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of **(pid_t)–1** is returned to the parent process, no child process is created, and **errno** is set to indicate the error.

ERRORS

fork( ) fails and no child process are created if one or more of the following is true:

EAGAIN There are two conditions that will cause an EAGAIN error.

The system-imposed limit on the total number of processes under

execution by a single user would be exceeded.

The total amount of system memory available is temporarily insufficient to duplicate this process.

ENOMEM There is not enough swap space.

SEE ALSO

alarm(2), **exec**(2), **exit**(2), **fcntl**(2), **getitimer**(2), **getrlimit**(2), **memcntl**(2), **mmap**(2), **nice**(2), **priocntl**(2), **ptrace**(2), **semop**(2), **shmop**(2), **times**(2), **umask**(2), **wait**(2), **exit**(3C), **plock**(3C), **pthread_atfork**(3T), **signal**(3C), **system**(3S), **thr_create**(3T)**, timer_create**(3R)

NOTES

Be careful to call **_exit( )** rather than **exit**(3C) if you cannot **execve( )**, since **exit**(3C) will flush and close standard I/O channels, and thereby corrupt the parent processes standard I/O data structures. Using **exit**(3C) will flush buffered data twice. See **exit**(2).

When calling **fork1( )** the thread (or LWP) in the child must not depend on any resources that are held by threads (or LWPs) that no longer exist in the child. In particular, locks held by these threads (or LWPs) will not be released.

In a multi-threaded process, **fork( )** or **fork1( )** can cause blocking system calls to be interrupted and return with an error of EINTR.

NAME

fpathconf, pathconf – get configurable pathname variables

SYNOPSIS

#include <unistd.h>

long fpathconf(int *fildes***, int** *name***);**

long pathconf(const char **path***, int** *name***);**

MT-LEVEL

pathconf( ) is Async-Signal-Safe

DESCRIPTION

The functions **fpathconf( )** and **pathconf( )** return the current value of a configurable limit or option associated with a file or directory. The *path* argument points to the pathname of a file or directory; *fildes* is an open file descriptor; and *name* is the symbolic constant (defined in **<unistd.h>**) representing the configurable system limit or option to be returned.

The values returned by **pathconf( )** and **fpathconf( )** depend on the type of file specified by *path* or *fildes*. The following table contains the symbolic constants supported by **pathconf( )** and **fpathconf( )** along with the POSIX defined return value. The return value is based on the type of file specified by *path* or *fildes*.

Value of *name*	See Note
_PC_LINK_MAX	1
_PC_MAX_CANNON	2
_PC_MAX_INPUT	2
_PC_NAME_MAX	3,4
_PC_PATH_MAX	4,5
_PC_PIPE_BUF	6
_PC_CHOWN_RESTRICTED	7
_PC_NO_TRUNC	3,4
_PC_VDISABLE	2
_PC_ASYNC_IO	2
_PC_PRIO_IO	2
_PC_SYNC_IO	1

Notes:

1 If *path* or *fildes* refers to a directory, the value returned applies to the directory itself.

2 The behavior is undefined if *path* or *fildes* does not refer to a terminal file.

3 If *path* or *fildes* refers to a directory, the value returned applies to the filenames within the directory.

4 The behavior is undefined if *path* or *fildes* does not refer to a directory.

5 If *path* or *fildes* 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* or *fildes* refers to a pipe or FIFO, the value returned applies to the pipe or FIFO. If *path* or *fildes* refers to a directory, the value returned applies to any FIFOs that exist or can be created within the directory. If *path* or *fildes* refer to any other type of file, the behavior is undefined.

7 If *path* or *fildes* refers to a directory, the value returned applies to any files, other than directories, that exist or can be created within the directory.

The value of the configurable system limit or option specified by *name* does not change during the lifetime of the calling process.

RETURN VALUES

If **fpathconf** or **pathconf** are invoked with an invalid symbolic constant or the symbolic constant corresponds to a configurable system limit or option not supported on the system, a value of –1 is returned to the invoking process. If the function fails because the configurable system limit or option corresponding to *name* is not supported on the system the value of **errno** is not changed.

ERRORS

fpathconf( ) fails if the following is true:

EBADF *fildes* is not a valid file descriptor.

pathconf( ) fails if one or more of the following are true:

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

ELOOP too many symbolic links are encountered while translating *path*.

EMULTIHOP components of *path* require hopping to multiple remote machines and file system type does not allow it.

ENAMETOOLONG the length of a pathname exceeds **{PATH_MAX}**, or a pathname component is longer than **{NAME_MAX}** while {**_POSIX_NO_TRUNC**} is in effect.

ENOENT *path* is needed for the command specified and the named file does not exist or if the *path* argument points to an empty string.

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

ENOTDIR a component of the path prefix is not a directory.

Both **fpathconf( )** and **pathconf( )** fail if the following is true:

EINVAL if *name* is an invalid value.

SEE ALSO

sysconf(3C), **limits**(4)

NAME

getaudit, setaudit – get and set process audit information

SYNOPSIS

cc [*flag* ...] *file* ... **–lbsm –lsocket –lnsl –lintl** [*library* ...]

```
#include <sys/param.h>
#include <bsm/audit.h>

int getaudit( struct auditinfo *info);

int setaudit( struct auditinfo *info);
```

AVAILABILITY

The functionality described in this man page is available only if the Basic Security Module (BSM) has been enabled. See **bsmconv**(1M) for more information.

DESCRIPTION

getaudit( ) gets the audit ID, the preselection mask, the terminal ID and the audit session ID of the current process.

setaudit( ) sets the audit ID, the preselection mask, the terminal ID and the audit session ID for the current process.

The **info** structure used to pass the process audit information contains the following members:

```
au_id_t      ai_auid;     /* audit user ID */
au_mask_t    ai_mask;     /* preselection mask */
au_tid_t     ai_termid;   /* terminal ID */
au_asid_t    ai_asid;     /* audit session ID */
```

Only processes with the effective user ID of the super-user may successfully execute these calls.

RETURN VALUES

getaudit( ) and **setaudit( )** return:

0 on success.

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

ERRORS

EFAULT The *info* parameter points outside the process's allocated address space.

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

SEE ALSO

audit(2)

NAME

getauid, setauid – get and set user audit identity

SYNOPSIS

cc [*flag* ...] *file* ... **–lbsm –lsocket –lnsl –lintl** [*library* ...]

```
#include <sys/param.h>
#include <bsm/audit.h>

int getauid( au_id_t *auid);

int setauid( au_id_t *auid);
```

AVAILABILITY

The functionality described in this man page is available only if the Basic Security Module (BSM) has been enabled. See **bsmconv**(1M) for more information.

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

EFAULT	*auid* points to an invalid address.
EPERM	The process's effective user ID is not super-user.

SEE ALSO

audit(2), **getaudit**(2)

NOTES

These system calls have been superseded by **getaudit( )** and **setaudit( )**.

NAME

getcontext, setcontext – get and set current user context

SYNOPSIS

```
#include <ucontext.h>
int getcontext(ucontext_t *ucp);
int setcontext(ucontext_t *ucp);
```

DESCRIPTION

These functions, along with those defined in **makecontext**(3C), are useful for implementing user level context switching between multiple threads of control within a process.

getcontext( ) initializes the structure pointed to by *ucp* to the current user context of the calling process. The user context is defined by **ucontext**(5) and includes the contents of the calling process's machine registers, signal mask and execution stack.

setcontext( ) restores the user context pointed to by *ucp*. The call to **setcontext( )** does not return; program execution resumes at the point specified by the context structure passed to **setcontext( ).** The context structure should have been one created either by a prior call to **getcontext( )** or **makecontext( )** or passed as the third argument to a signal handler (see **sigaction**(2)). If the context structure was one created with **getcontext( ),** program execution continues as if the corresponding call of **getcontext( )** had just returned. If the context structure was one created with **makecontext**, program execution continues with the function specified to **makecontext**.

RETURN VALUES

On successful completion, **setcontext( )** does not return and **getcontext( )** returns 0. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

ERRORS

getcontext( ) and **setcontext( )** will fail if the following is true:

EFAULT *ucp* points to an illegal address.

SEE ALSO

sigaction(2), **sigaltstack**(2), **sigprocmask**(2), **makecontext**(3C), **ucontext**(5)

NOTES

When a signal handler is executed, the current user context is saved and a new context is created by the kernel. If the process leaves the signal handler via **longjmp**(3C) the original context will not be restored, and future calls to **getcontext( )** will not be reliable. Signal handlers should use **siglongjmp**(3C) or **setcontext( )** instead.

NAME

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

SYNOPSIS

```
#include <sys/dirent.h>
int getdents(int fildes, struct dirent *buf, size_t nbyte);
```

DESCRIPTION

getdents() attempts to read *nbyte* bytes from the directory associated with the file descriptor *fildes* and to format them as file system independent directory entries in the buffer pointed to by *buf*. Since the file system independent directory entries are of variable length, in most cases the actual number of bytes returned will be strictly less than *nbyte*. See **dirent**(4) to calculate the number of bytes.

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

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

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

RETURN VALUES

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

ERRORS

getdents() will fail if one or more of the following are true:

EBADF	*fildes* is not a valid file descriptor open for reading.
EFAULT	*buf* points to an illegal address.
EINVAL	*nbyte* is not large enough for one directory entry.
EIO	An I/O error occurred while accessing the file system.
ENOENT	The current file pointer for the directory is not located at a valid entry.
ENOLINK	*fildes* points to a remote machine and the link to that machine is no longer active.
ENOTDIR	*fildes* is not a directory.

SEE ALSO

directory(3C), **dirent**(4)

NAME

getgroups, setgroups – get or set supplementary group access list IDs

SYNOPSIS

```
#include <unistd.h>
int getgroups(int gidsetsize, gid_t *grouplist);
int setgroups(int ngroups, const gid_t *grouplist);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

getgroups() gets the current supplemental group access list of the calling process and stores the result in the array of group IDs specified by *grouplist*. This array has *gidsetsize* entries and must be large enough to contain the entire list. This list cannot be greater than **NGROUPS_MAX**. If *gidsetsize* equals 0, **getgroups()** will return the number of groups to which the calling process belongs without modifying the array pointed to by *grouplist*.

setgroups() sets the supplementary group access list of the calling process from the array of group IDs specified by *grouplist*. The number of entries is specified by *ngroups* and can not be greater than **NGROUPS_MAX**. This function may be invoked only by the super-user.

RETURN VALUES

Upon successful completion, **getgroups()** returns the number of supplementary group IDs set for the calling process and **setgroups()** returns the value 0. Otherwise, a value of –1 is returned and **errno** is set to indicate the error.

ERRORS

getgroups() will fail if:

EINVAL The value of *gidsetsize* is non-zero and less than the number of supplementary group IDs set for the calling process.

setgroups() will fail if:

EINVAL The value of *ngroups* is greater than **NGROUPS_MAX**.

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

Either call will fail if:

EFAULT A referenced part of the array pointed to by *grouplist* is an illegal address.

SEE ALSO

groups(1), **chown**(2), **getuid**(2), **setuid**(2), **getgrnam**(3C), **initgroups**(3C)

NAME

getitimer, setitimer – get or set value of interval timer

SYNOPSIS

```
#include <sys/time.h>
int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);
```

MT-LEVEL

MT-Safe

DESCRIPTION

The system provides each process with four interval timers, defined in **sys/time.h**. The **getitimer( )** function 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 **NULL**, stores the previous value of the timer in the structure pointed to by *ovalue*.

A timer value is defined by the **itimerval** structure (see **gettimeofday**(3C) for the definition of **timeval**), which includes the following members:

```
struct timeval    it_interval;    /* timer interval */
struct timeval    it_value;       /* current value */
```

it_value indicates the time to the next timer expiration. **it_interval** specifies a value to be used in reloading **it_value** when the timer expires. Setting **it_value** to zero disables a timer, regardless of the value of **it_interval**. Setting **it_interval** to zero disables a timer after its next expiration (assuming **it_value** is non-zero).

Time values smaller than the resolution of the system clock are rounded up to the resolution of the system clock, except for **ITIMER_REALPROF,** whose values are rounded up to the resolution of the profiling clock.

The four timers are:

ITIMER_REAL Decrements in real time. A **SIGALRM** signal is delivered when this timer expires.

In the current and previous releases, when **setitimer(ITIMER_REAL, ...)** is called in a multi-thread process linked with **-lthread** (Solaris threads) or **-lpthread** (POSIX threads), the resulting **SIGALRM** is sent to the bound thread that called **setitimer( )**, i.e. **setitimer( )** has a per-thread semantic when called from a bound thread. This semantic will become obsolete in a future release. The semantic will move to a per-process semantic, i.e. the resulting **SIGALRM** will be sent to the process. The **SIGALRM** so generated is not maskable on this bound thread via any signal masking function, **pthread_sigmask**(3T), **thr_sigsetmask**(3T), or **sigprocmask**(2). This is a bug that will not be fixed, since the per-thread semantic will be discontinued in the next release.

Also, calling this routine from an unbound thread is not guaranteed to work as in the case of bound threads. The

resulting **SIGALRM** may be sent to some other thread (see **alarm**(2)). This is a bug and will not be fixed since the per-thread semantic is going to be discontinued.

Calling **setitimer(ITIMER_REAL, ...)** from a process linked with **-lpthread** (POSIX threads) has the same behavior as Solaris threads described above, where a Solaris bound thread is the same as a POSIX thread in system scheduling scope and a Solaris unbound thread is the same as a POSIX thread in local scheduling scope.

Hence, for multi-threaded (Solaris or POSIX) programs in the current and previous releases, the only reliable way to use the **ITIMER_REAL** flag is to call it from a bound thread which does not mask **SIGALRM** and to expect the **SIGALRM** to be delivered to this bound thread.

The current working of this flag is not being improved since some applications might depend on the current (slightly broken) semantic. When this semantic is discontinued in the future, it will be replaced with a per-process semantic, i.e. using this flag from any thread, bound or unbound, will result in the **SIGALRM** being sent to the process.

New MT applications should not use this flag, and should use **alarm**(2) instead.

ITIMER_VIRTUAL Decrements in process virtual time. It runs only when the process is executing. A **SIGVTALRM** signal is delivered when it expires. (For multi-threaded programs see ''Warnings'' section below).

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 functions, programs using this timer must be prepared to restart interrupted functions. (For multi-threaded programs see ''Warnings'' section below).

ITIMER_REALPROF Decrements in real time. It is designed to be used for real-time profiling of multithreaded programs. Each time the **ITIMER_REALPROF** timer expires, one counter in a set of counters maintained by the system for each lightweight process (lwp) is incremented. The counter corresponds to the state of the lwp at the time of the timer tick. All lwps executing in user mode when the timer expires are interrupted into system mode. When each lwp resumes execution in user mode, if any of the elements in its set of counters are non-zero, the **SIGPROF** signal

is delivered to the lwp. The **SIGPROF** signal is delivered before any other signal except **SIGKILL.** This signal does not interrupt any in-progress function. A **siginfo** structure, defined in **sys/siginfo.h**, is associated with the delivery of the **SIGPROF** signal, and includes the following members:

```
si_tstamp;      /* high resolution timestamp */
si_syscall;     /* current syscall */
si_nsysarg;     /* number of syscall arguments */
si_sysarg[ ];   /* actual syscall arguments */
si_fault;       /* last fault type */
si_faddr;       /* last fault address */
si_mstate[ ];   /* ticks in each microstate */
```

The enumeration of microstates (indices into **si_mstate)** is defined in **sys/msacct.h**. (For multi-threaded programs see ''Warnings'' section below).

RETURN VALUES

If the calls succeed, a value of 0 is returned. If an error occurs, the value –1 is returned, and an error code is placed in the global variable **errno.**

ERRORS

getitimer( ) and **setitimer( )** will fail if:

EINVAL The specified number of seconds is greater than 100,000,000, the number of microseconds is greater than or equal to 1,000,000, or the *which* parameter is unrecognized.

setitimer( ) will fail if:

EACCES An unbound Solaris thread or a POSIX thread in local scheduling scope, with a flag other than **ITIMER_REAL,** called **setitimer( )**.

SEE ALSO

alarm(2), **sigprocmask**(2), **gettimeofday**(3C), **sysconf**(3C), **pthread_attr_setscope**(3T), **pthread_sigmask**(3T)

WARNINGS

All flags to **setitimer( )**, other than **ITIMER_REAL,** only behave as documented with ''bound'' threads. Additionally, their ability to mask the signal only works with bound threads. If the call is made using one of these flags from an unbound thread, the system call returns **-1** and sets **errno** to **EACCES.**

These behaviors are the same for bound or unbound POSIX threads. (A POSIX thread with system-wide scope, created by the call

pthread_attr_setscope(&attr**, PTHREAD_SCOPE_SYSTEM);**

is equivalent to a Solaris bound thread. A POSIX thread with local process scope, created by the call

pthread_attr_setscope(&attr**, PTHREAD_SCOPE_PROCESS);**

is equivalent to a Solaris unbound thread.

NOTES

The microseconds field should not be equal to or greater than one second.

setitimer() is independent of the **alarm()** function.

Do not use **setitimer(ITIMER_REAL)** with the **sleep()** routine. A **sleep()** wipes out knowledge of the user signal handler for **SIGALRM.**

ITIMER_PROF and **ITIMER_REALPROF** deliver the same signal and have different semantics. They cannot be used together.

The granularity of the resolution of alarm time is platform-dependent.

NAME

getmsg, getpmsg – get next message off a stream

SYNOPSIS

```
#include <stropts.h>
int getmsg(int fildes, struct strbuf *ctlptr, struct strbuf *dataptr, int *flagsp);
int getpmsg(int fildes, struct strbuf *ctlptr, struct strbuf *dataptr, int *bandp, int *flagsp);
```

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.

The function **getpmsg()** does the same thing as **getmsg()**, but provides finer control over the priority of the messages received. Except where noted, all information pertaining to **getmsg()** also pertains to **getpmsg()**.

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

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

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 0 if there is a zero-length control or data part, or -1 if no data or control information is present in the message. *flagsp* should point to an integer that indicates the type of message the user is able to receive. This is described later.

ctlptr is used to hold the control part from the message and *dataptr* is used to hold the data part from the message. If *ctlptr* (or *dataptr*) is **NULL** or the **maxlen** field is –1, the control (or data) part of the message is not processed and is left on the stream head read queue. If *ctlptr* (or *dataptr*) is not **NULL** and there is no corresponding control (or data) part of the messages on the stream head read queue, **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.

By default, **getmsg()** processes the first available message on the stream head read queue. However, a user may choose to retrieve only high priority messages by setting the integer pointed by *flagsp* to **RS_HIPRI**. In this case, **getmsg()** processes the next message only if it is a high priority message.

If the integer pointed by *flagsp* is 0, **getmsg()** retrieves any message available on the stream head read queue. In this case, on return, the integer pointed to by *flagsp* will be set to **RS_HIPRI** if a high priority message was retrieved, or 0 otherwise.

For **getpmsg()**, the flags are different. *flagsp* points to a bitmask with the following mutually-exclusive flags defined: **MSG_HIPRI, MSG_BAND**, and **MSG_ANY**. Like **getmsg()**, **getpmsg()** processes the first available message on the stream head read queue. A user may choose to retrieve only high-priority messages by setting the integer pointed to by *flagsp* to **MSG_HIPRI** and the integer pointed to by *bandp* to 0. In this case, **getpmsg()** will only process the next message if it is a high-priority message. In a similar manner, a user may choose to retrieve a message from a particular priority band by setting the integer pointed to by *flagsp* to **MSG_BAND** and the integer pointed to by *bandp* to the priority band of interest. In this case, **getpmsg()** will only process the next message if it is in a priority band equal to, or greater than, the integer pointed to by *bandp*, or if it is a high-priority message. If a user just wants to get the first message off the queue, the integer pointed to by *flagsp* should be set to **MSG_ANY** and the integer pointed to by *bandp* should be set to 0. On return, if the message retrieved was a high-priority message, the integer pointed to by *flagsp* will be set to **MSG_HIPRI** and the integer pointed to by *bandp* will be set to 0. Otherwise, the integer pointed to by *flagsp* will be set to **MSG_BAND** and the integer pointed to by *bandp* will be set to the priority band of the message.

If **O_NDELAY** and **O_NONBLOCK** are clear, **getmsg()** blocks until a message of the type specified by *flagsp* is available on the stream head read queue. If **O_NDELAY** or **O_NONBLOCK** has been set and a message of the specified type 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()** continues to operate normally, as described above, until the stream head read queue is empty. Thereafter, it returns 0 in the **len** fields of *ctlptr* and *dataptr*.

RETURN VALUES

Upon successful completion, a non-negative value is returned. A value of 0 indicates that a full message was read successfully. A return value of **MORECTL** indicates that more control information is waiting for retrieval. A return value of **MOREDATA** indicates that more data are waiting for retrieval. A return value of **MORECTL | MOREDATA** indicates that both types of information remain. Subsequent **getmsg** calls retrieve the remainder of the message. However, if a message of higher priority has come in on the stream head read queue, the next call to **getmsg** will retrieve that higher priority message before retrieving the remainder of the previously received partial message.

ERRORS

getmsg() or **getpmsg()** will fail if one or more of the following are true:

EAGAIN The **O_NDELAY** or **O_NONBLOCK** flag is set, and no messages are available.

EBADF *fildes* is not a valid file descriptor open for reading.

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

EFAULT *ctlptr*, *dataptr*, *bandp*, or *flagsp* points to an illegal address.

EINTR A signal was caught during the **getmsg** function.

EINVAL An illegal value was specified in *flagsp*, or the stream referenced by *fildes* is linked under a multiplexor.

ENOSTR A stream is not associated with *fildes*.

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**(2), **write**(2)

STREAMS Programming Guide

NAME

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

SYNOPSIS

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

pid_t getpid(void);

pid_t getpgrp(void);

pid_t getppid(void);

pid_t getpgid(pid_t pid);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

getpid( ) returns the process ID of the calling process.

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

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

getpgid( ) returns the process group ID of the process whose process ID is equal to *pid,* or the process group ID of the calling process, if *pid* is equal to zero.

RETURN VALUES

Upon successful completion, all return the process group ID. On failure, **getpgid( ) returns a value of** (**pid_t**) –1 and sets **errno** to indicate the error.

ERRORS

getpgid( ) will fail if one or more of the following is true:

EPERM The process whose process ID is equal to *pid* is not in the same session as the calling process, and the implementation does not allow access to the process group ID of that process from the calling process.

ESRCH There is no process with a process ID equal to *pid.*

SEE ALSO

intro(2), **exec**(2), **fork**(2), **getsid**(2), **setpgid**(2), **setpgrp**(2), **signal**(3C)

NAME

getrlimit, setrlimit – control maximum system resource consumption

SYNOPSIS

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

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

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

DESCRIPTION

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

Each call to either **getrlimit( )** or **setrlimit( )** identifies a specific resource to be operated upon as well as a resource limit. A resource limit is a pair of values: one specifying the current (soft) limit, the other a maximum (hard) limit. Soft limits may be changed by a process to any value that is less than or equal to the hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or equal to the soft limit. Only a process with an effective user ID of super-user can raise a hard limit. Both hard and soft limits can be changed in a single call to **setrlimit( )** subject to the constraints described above. Limits may have an "infinite" value of **RLIM_INFINITY**. *rlp* is a pointer to **struct rlimit** that includes the following members:

```
rlim_t    rlim_cur;    /* current (soft) limit */
rlim_t    rlim_max;    /* hard limit */
```

rlim_t is an arithmetic data type to which objects of type **int**, **size_t**, and **off_t** can be cast without loss of information.

The possible resources, their descriptions, and the actions taken when the current limit is exceeded are summarized in the table below:

RLIMIT_CORE The maximum size of a core file in bytes that may be created by a process. A limit of **0** will prevent the creation of a core file.

The writing of a core file will terminate at this size.

RLIMIT_CPU The maximum amount of CPU time in seconds used by a process. This is a soft limit only.

SIGXCPU is sent to the process. If the process is holding or ignoring **SIGXCPU**, the behavior is scheduling class defined.

RLIMIT_DATA The maximum size of a process's heap in bytes.

brk(2) will fail with **errno** set to **ENOMEM.**

RLIMIT_FSIZE The maximum size of a file in bytes that may be created by a process. A limit of **0** will prevent the creation of a file.

SIGXFSZ is sent to the process. If the process is holding or ignoring **SIGXFSZ**, continued attempts to increase the size of a file beyond the limit will fail with **errno** set to **EFBIG.**

RLIMIT_NOFILE One more than the maximum value that the system may assign to a newly created descriptor. This limit constrains the number of file descriptors that a process may create.

RLIMIT_STACK The maximum size of a process's stack in bytes. The system will not automatically grow the stack beyond this limit.

SIGSEGV is sent to the process. If the process is holding or ignoring **SIGSEGV**, or is catching **SIGSEGV** and has not made arrangements to use an alternate stack (see **sigaltstack**(2)), the disposition of **SIGSEGV** will be set to **SIG_DFL** before it is sent.

RLIMIT_VMEM The maximum size of a process's mapped address space in bytes.

brk(2) and **mmap**(2) functions will fail with **errno** set to **ENOMEM.** In addition, the automatic stack growth will fail with the effects outlined above.

Because limit information is stored in the per-process information, the shell builtin **ulimit** command must directly execute this system call if it is to affect all future processes created by the shell.

The value of the current limit of the following resources affect these implementation defined parameters:

Limit	Implementation Defined Constant
RLIMIT_FSIZE	**FCHR_MAX**
RLIMIT_NOFILE	**OPEN_MAX**

RETURN VALUES

Upon successful completion, the function **getrlimit( )** returns a value of 0; otherwise, it returns a value of –1 and sets **errno** to indicate an error.

ERRORS

Under the following conditions, the functions **getrlimit( )** and **setrlimit( )** fail and set **errno** to:

EFAULT *rlp* points to an illegal address.

EINVAL An invalid *resource* was specified; or in a **setrlimit( )** call, the new **rlim_cur** exceeds the new **rlim_max**.

EPERM The limit specified to **setrlimit( )** would have raised the maximum limit value, and the effective user of the calling process is not super-user.

SEE ALSO

brk(2), **open**(2), **sigaltstack**(2), **malloc**(3C), **signal**(3C), **signal**(5)

NOTES

RLIMIT_STACK:

Within a process **setrlimit()**, will increase the limit on the size of your stack, but will not move current memory segments to allow for that growth. Therefore, to guarantee that the process stack can grow to the limit, you must alter the limit prior to the execution of the process in which the new stack size is to be used.

NAME

getsid, setsid – get or set session ID

SYNOPSIS

```
#include <sys/types.h>
pid_t getsid(pid_t pid);
#include <sys/types.h>
#include <unistd.h>
pid_t setsid(void);
```

MT-LEVEL

setsid( ) is Async-Signal-Safe

DESCRIPTION

The function **getsid( )** returns the session ID of the process whose process ID is equal to *pid*. If *pid* is equal to **(pid_t)0**, **getsid( )** returns the session ID of the calling process.

If the calling process is not already a process group leader, **setsid( )** sets the process group ID and session ID of the calling process to the process ID of the calling process, and releases the process's controlling terminal.

See **intro**(2) for more information on process groups and controlling terminals.

RETURN VALUES

Upon successful completion, **getsid( )** and **setsid( )** return the session ID of the specified process. Otherwise, **getsid( )** returns a value of **(pid_t)–1** and sets **errno** to indicate an error, and **setsid( )** returns a value of –1 and sets **errno** to indicate the error.

ERRORS

Under the following conditions, **getsid( )** fails and sets **errno** to:

EPERM The process whose process ID is equal to *pid* is not in the same session as the calling process, and the implementation does not allow access to the session ID of that process from the calling process.

ESRCH There is no process with a process ID equal to *pid*.

setsid( ) will fail and return an error if the following is true:

EPERM The calling process is already a process group leader, or there are processes other than the calling process whose process group ID is equal to the process ID of the calling process.

SEE ALSO

intro(2), **exec**(2), **fork**(2), **getpid**(2), **setpgid**(2)

WARNINGS

A call to **setsid( )** by a process that is a process group leader will fail. A process can become a process group leader by being the last member of a pipeline started by a job control shell. Thus, a process that expects to be part of a pipeline, and that calls **setsid( )**, should always first fork; the parent should exit and the child should call **setsid( )**. This will ensure that the calling process will work reliably when started by both job control shells and non-job control shells.

NAME

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

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

getuid( ) returns the real user ID of the calling process. The real user ID identifies the person who is logged in.

geteuid( ) returns the effective user ID of the calling process. The effective user ID gives the process various permissions during execution of ''set-user-ID'' mode processes which use **getuid ( )** to determine the real user ID of the process that invoked them.

getgid( ) returns the real group ID of the calling process.

getegid( ) returns the effective group ID of the calling process.

SEE ALSO

intro(2), **setuid**(2)

NAME

ioctl – control device

SYNOPSIS

```
#include <unistd.h>
int ioctl(int fildes, int request, /* arg */ ...);
```

DESCRIPTION

ioctl( ) performs a variety of control functions on devices and STREAMS. For non-STREAMS files, the functions performed by this call are device-specific control functions. *request* and an optional third argument with varying type are passed to the file designated by *fildes* and are interpreted by the device driver.

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

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

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

RETURN VALUES

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

ERRORS

ioctl( ) fails for any type of file if one or more of the following are true:

EBADF *fildes* is not a valid open file descriptor.

EINTR A signal was caught during the **ioctl( )** function.

ENOTTY *fildes* is not associated with a device driver that accepts control functions.

ioctl( ) also fails if the device driver detects an error. In this case, the error is passed through **ioctl( )** without change to the caller. A particular driver might not have all of the following error cases. Under the following conditions, requests to device drivers may fail and set **errno** to:

EFAULT
request requires a data transfer to or from a buffer pointed to by *arg*, but *arg* points to an illegal address.

EINVAL
request or *arg* is not valid for this device.

EIO Some physical I/O error has occurred.

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

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

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

SEE ALSO **streamio**(7I), **termio**(7I)

NAME

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

SYNOPSIS

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

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

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

kill( ) sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal that is to be sent is specified by *sig* and is either one from the list given in **signal** (see **signal**(5)), or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The real or effective user ID of the sending process must match the real or saved (from **exec**(2)) user ID of the receiving process unless the effective user ID of the sending process is super-user, (see **intro**(2)), or *sig* is **SIGCONT** and the sending process has the same session ID as the receiving process.

If *pid* is greater than 0, *sig* will be sent to the process whose process ID is equal to *pid*.

If *pid* is negative but not **(pid_t)–1**, *sig* will be sent to all processes whose process group ID is equal to the absolute value of *pid* and for which the process has permission to send a signal.

If *pid* is **0**, *sig* will be sent to all processes excluding special processes (see **intro**(2)) whose process group ID is equal to the process group ID of the sender.

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

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

RETURN VALUES

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

ERRORS

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

EINVAL *sig* is not a valid signal number.

EPERM *sig* is **SIGKILL** and *pid* is **(pid_t)1** (that is, the calling process does not have permission to send the signal to any of the processes specified by *pid*).

EPERM The effective user of the calling process does not match the real or saved user and is not super-user, and the calling process is not sending **SIGCONT** to a process that shares the same session ID.

ESRCH No process or process group can be found corresponding to that specified by *pid*.

SEE ALSO **kill**(1), **intro**(2), **exec**(2), **getpid**(2), **getsid**(2), **setpgrp**(2), **sigaction**(2), **sigsend**(2), **signal**(3C), **signal**(5)

NOTES **sigsend**(2) is a more versatile way to send signals to processes.

NAME

link – link to a file

SYNOPSIS

#include <unistd.h>

int link(const char **existing*, **const char** **new*);

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

link() creates a new link (directory entry) for the existing file and increments its link count by one. *existing* points to a path name naming an existing file. *new* points to a path name naming the new directory entry to be created.

To create 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 *existing* 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

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

ERRORS

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

EACCES	A component of either path prefix denies search permission.
EACCES	The requested link requires writing in a directory with a mode that denies write permission.
EEXIST	The link named by *new* exists.
EFAULT	*existing* or *new* points to an illegal address.
EINTR	A signal was caught during the **link()** function.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMLINK	The maximum number of links to a file would be exceeded.
EMULTIHOP	Components of *existing* or *new* require hopping to multiple remote machines and the file system type does not allow it.
ENAMETOOLONG	The length of the *existing* or *new* argument exceeds {**PATH_MAX**}, or the length of a *existing* or *new* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	*existing* or *new* is a null path name.
ENOENT	A component of either path prefix does not exist.
ENOENT	The file named by *existing* does not exist.
ENOLINK	*existing* or *new* points to a remote machine and the link to that machine is no longer active.

ENOSPC	the directory that would contain the link cannot be extended.
ENOTDIR	A component of either path prefix is not a directory.
EPERM	The file named by *existing* is a directory and the effective user of the calling process is not super-user.
EROFS	The requested link requires writing in a directory on a read-only file system.
EXDEV	The link named by *new* and the file named by *existing* are on different logical devices (file systems).

SEE ALSO **symlink**(2), **unlink**(2)

NAME

llseek – move extended read / write file pointer

SYNOPSIS

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

offset_t llseek(int fildes, offset_t offset, int whence);
```

DESCRIPTION

llseek( ) sets the 64-bit extended file pointer associated with the open file descriptor specified by *fildes* as follows:

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

On success, **llseek( )** returns the resulting pointer location, as measured in bytes from the beginning of the file.

RETURN VALUES

Upon successful completion, the resulting file pointer is returned. Remote file descriptors are the only ones that allow negative file pointers. Otherwise, a value of –1 is returned and **errno** is set to indicate the error.

ERRORS

llseek( ) fails and the file pointer remains unchanged if one or more of the following are true:

EBADF	*fildes* is not an open file descriptor.
EINVAL	*whence* is not **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**.
EINVAL	*offset* is not a valid offset for this file system type.
EINVAL	*fildes* is not a remote file descriptor, and the resulting file pointer would be negative.
ESPIPE	*fildes* is associated with a pipe or fifo.

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

LIMITATIONS

Although each file has a 64-bit file pointer associated with it, existing file system types do not support the full range of 64-bit offsets. In particular, non-device files remain limited to offsets of less than two gigabytes. Device drivers may support offsets of up to 1024 gigabytes for device special files.

SEE ALSO

creat(2), **dup**(2), **fcntl**(2), **lseek**(2), **open**(2)

NAME

lseek – move read/write file pointer

SYNOPSIS

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

off_t lseek(int fildes, off_t offset, int whence);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

lseek() sets the file pointer associated with the open file descriptor specified by *fildes* as follows:

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

On success, **lseek()** returns the resulting pointer location, as measured in bytes from the beginning of the file. Note that if *fildes* is a remote file descriptor and *offset* is negative, **lseek()** returns the file pointer even if it is negative.

lseek() allows the file pointer to be set beyond the existing data in the file. If data are later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return bytes of value 0 until data are written into the gap.

RETURN VALUES

Upon successful completion, the resulting file pointer is returned. Remote file descriptors are the only ones that allow negative file pointers. Otherwise, a value of –1 is returned and **errno** is set to indicate the error.

ERRORS

lseek() fails and the file pointer remains unchanged if one or more of the following are true:

EBADF	*fildes* is not an open file descriptor.
EINVAL	*whence* is not **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**.
EINVAL	*fildes* is not a remote file descriptor, and the resulting file pointer would be negative.
ESPIPE	*fildes* is associated with a pipe or fifo.

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

SEE ALSO

creat(2), **dup**(2), **fcntl**(2), **open**(2), **read**(2), **write**(2)

NOTES

In multithreaded programs, using **lseek()** in conjunction with a **read()** or **write()** on a file descriptor shared amongst more than one thread is not an atomic operation. To ensure atomicity, use **pread()** or **pwrite()**.

NAME

memcntl – memory management control

SYNOPSIS

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

int memcntl(caddr_t addr, size_t len, int cmd, caddr_t arg, int attr, int mask);
```

MT-LEVEL

MT-Safe

DESCRIPTION

The function **memcntl( )** allows the calling process to apply a variety of control operations over the address space identified by the mappings established for the address range [*addr*, *addr* + *len*).

addr must be a multiple of the pagesize as returned by **sysconf**(3C). The scope of the control operations can be further defined with additional selection criteria (in the form of attributes) according to the bit pattern contained in *attr*.

The following attributes specify page mapping selection criteria:

SHARED Page is mapped shared.

PRIVATE Page is mapped private.

The following attributes specify page protection selection criteria:

PROT_READ Page can be read.

PROT_WRITE Page can be written.

PROT_EXEC Page can be executed.

The selection criteria are constructed by an OR of the attribute bits and must match exactly.

In addition, the following criteria may be specified:

PROC_TEXT Process text.

PROC_DATA Process data.

where **PROC_TEXT** specifies all privately mapped segments with read and execute permission, and **PROC_DATA** specifies all privately mapped segments with write permission.

Selection criteria can be used to describe various abstract memory objects within the address space on which to operate. If an operation shall not be constrained by the selection criteria, *attr* must have the value **0**.

The operation to be performed is identified by the argument *cmd*. The symbolic names for the operations are defined in **<sys/mman.h>** as follows:

MC_LOCK Lock in memory all pages in the range with attributes *attr*. A given page may be locked multiple times through different mappings; however, within a given mapping, page locks do not nest. Multiple lock operations on the same address in the same process will all be removed with a single unlock operation. A page locked in one process and mapped in another (or visible

through a different mapping in the locking process) is locked in memory as long as the locking process does neither an implicit nor explicit unlock operation. If a locked mapping is removed, or a page is deleted through file removal or truncation, an unlock operation is implicitly performed. If a writable **MAP_PRIVATE** page in the address range is changed, the lock will be transferred to the private page.

At present *arg* is unused, but must be **0** to ensure compatibility with potential future enhancements.

MC_LOCKAS Lock in memory all pages mapped by the address space with attributes *attr*. At present *addr* and *len* are unused, but must be **NULL** and **0** respectively, to ensure compatibility with potential future enhancements. *arg* is a bit pattern built from the flags:

MCL_CURRENT	Lock current mappings
MCL_FUTURE	Lock future mappings

The value of *arg* determines whether the pages to be locked are those currently mapped by the address space, those that will be mapped in the future, or both. If **MCL_FUTURE** is specified, then all mappings subsequently added to the address space will be locked, provided sufficient memory is available.

MC_SYNC Write to their backing storage locations all modified pages in the range with attributes *attr*. Optionally, invalidate cache copies. The backing storage for a modified **MAP_SHARED** mapping is the file the page is mapped to; the backing storage for a modified **MAP_PRIVATE** mapping is its swap area. *arg* is a bit pattern built from the flags used to control the behavior of the operation:

MS_ASYNC	perform asynchronous writes
MS_SYNC	perform synchronous writes
MS_INVALIDATE	invalidate mappings

MS_ASYNC returns immediately once all write operations are scheduled; with **MS_SYNC** the function will not return until all write operations are completed.

MS_INVALIDATE invalidates all cached copies of data in memory, so that further references to the pages will be obtained by the system from their backing storage locations. This operation should be used by applications that require a memory object to be in a known state.

MC_UNLOCK Unlock all pages in the range with attributes *attr*. At present *arg* is unused, but must be **0** to ensure compatibility with potential future enhancements.

MC_UNLOCKAS
Remove address space memory locks, and locks on all pages in the address space with attributes *attr*. At present *addr*, *len*, and *arg* are unused, but must be **NULL**, **0** and **0** respectively, to ensure compatibility with potential future enhancements.

The *mask* argument must be zero; it is reserved for future use.

Locks established with the lock operations are not inherited by a child process after **fork()**. **memcntl()** fails if it attempts to lock more memory than a system-specific limit.

Due to the potential impact on system resources, all operations, with the exception of **MC_SYNC**, are restricted to processes with super-user effective user ID . The **memcntl()** function subsumes the operations of **plock** and **mctl**.

RETURN VALUES

Upon successful completion, the function **memcntl()** returns a value of **0**; otherwise, it returns a value of **–1** and sets **errno** to indicate an error.

ERRORS

Under the following conditions, the function **memcntl()** fails and sets **errno** to:

EAGAIN if some or all of the memory identified by the operation could not be locked when **MC_LOCK** or **MC_LOCKAS** is specified.

EBUSY if some or all the addresses in the range [*addr*, *addr* + *len*) are locked and **MC_SYNC** with **MS_INVALIDATE** option is specified.

EINVAL if *addr* is not a multiple of the page size as returned by **sysconf**.

EINVAL if *addr* and / or *len* do not have the value **0** when **MC_LOCKAS** or **MC_UNLOCKAS** is specified.

EINVAL if *arg* is not valid for the function specified.

EINVAL if invalid selection criteria are specified in *attr*.

ENOMEM if some or all the addresses in the range [*addr*, *addr* + *len*) are invalid for the address space of the process or pages not mapped are specified.

EPERM if the process's effective user ID is not super-user and one of **MC_LOCK**, **MC_LOCKAS**, **MC_UNLOCK**, **MC_UNLOCKAS** was specified.

SEE ALSO

mmap(2), **mprotect**(2), **plock**(3C), **mlock**(3C), **mlockall**(3C), **msync**(3C), **sysconf**(3C)

NAME

mincore – determine residency of memory pages

SYNOPSIS

```
#include <sys/types.h>
int mincore(caddr_t addr, size_t len, char *vec);
```

DESCRIPTION

mincore( ) determines the residency of the memory pages in the address space covered by mappings in the range [*addr*, *addr* + *len*]. The status is returned as a character-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 are undefined and may contain other information in future implementations.

Because the status of a page can change after **mincore( )** checks it, but before **mincore( )** returns the information, returned information might be outdated. Only locked pages are guaranteed to remain in memory; see **mlock**(3C).

RETURN VALUES

mincore( ) returns 0 on success, –1 on failure and sets **errno** to indicate the error.

ERRORS

mincore( ) fails if:

EFAULT	*vec* points to an illegal address.
EINVAL	*addr* is not a multiple of the page size as returned by **sysconf**(3C).
EINVAL	The argument *len* has a value less than or equal to **0**.
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), **mlock**(3C), **sysconf**(3C)

NAME

mkdir – make a directory

SYNOPSIS

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

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

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

mkdir( ) creates a new directory named by the path name pointed to by *path*. The mode of the new directory is initialized from *mode* (see **chmod**(2) for values of mode). The protection part of the *mode* argument is modified by the process's file creation mask (see **umask**(2)).

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to the process's effective group ID, or if the **S_ISGID** bit is set in the parent directory, then the group ID of the directory is inherited from the parent. The **S_ISGID** bit of the new directory is inherited from the parent directory.

If *path* is a symbolic link, it is not followed.

The newly created directory is empty with the exception of entries for itself (**.**) and its parent directory (**..**).

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

RETURN VALUES

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

ERRORS

mkdir( ) fails and creates no directory if one or more of the following are true:

EACCES	Either a component of the path prefix denies search permission or write permission is denied on the parent directory of the directory to be created.
EEXIST	The named file already exists.
EFAULT	*path* points to an illegal address.
EIO	An I/O error has occurred while accessing the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMLINK	The maximum number of links to the parent directory would be exceeded.
EMULTIHOP	Components of *path* require hopping to multiple remote machines and the file system type does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.

ENOENT	A component of the path prefix does not exist or is a null pathname.
ENOLINK	*path* points to a remote machine and the link to that machine is no longer active.
ENOSPC	No free space is available on the device containing the directory.
ENOTDIR	A component of the path prefix is not a directory.
EROFS	The path prefix resides on a read-only file system.

SEE ALSO **chmod**(2), **mknod**(2), **umask**(2), **stat**(5)

NAME

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

SYNOPSIS

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

int mknod(const char *path, mode_t mode, dev_t dev);
```

DESCRIPTION

mknod() creates a new file named by the path name pointed to by *path*. The file type and permissions of the new file are initialized from *mode*.

The file type is specified in *mode* by the **S_IFMT** bits, which must be set to one of the following values:

S_IFIFO	fifo special
S_IFCHR	character special
S_IFDIR	directory
S_IFBLK	block special
S_IFREG	ordinary file

The file access permissions are specified in *mode* by the 0007777 bits, and may be constructed by an OR of the following values:

S_ISUID	04000	Set user ID on execution.
S_ISGID	020#0	Set group ID on execution if # is **7**, **5**, **3**, or **1**. Enable mandatory file/record locking if # is **6**, **4**, **2**, or **0**.
S_ISVTX	01000	Save text image after execution.
S_IRWXU	00700	Read, write, execute by owner.
S_IRUSR	00400	Read by owner.
S_IWUSR	00200	Write by owner.
S_IXUSR	00100	Execute (search if a directory) by owner.
S_IRWXG	00070	Read, write, execute by group.
S_IRGRP	00040	Read by group.
S_IWGRP	00020	Write by group.
S_IXGRP	00010	Execute by group.
S_IRWXO	00007	Read, write, execute (search) by others.
S_IROTH	00004	Read by others.
S_IWOTH	00002	Write by others
S_IXOTH	00001	Execute by others.

The owner ID of the file is set to the effective user ID of the process. The group ID of the file is set to the effective group ID of the process. However, if the **S_ISGID** bit is set in the parent directory, then the group ID of the file is inherited from the parent. If the group ID of the new file does not match the effective group ID or one of the supplementary group IDs, the **S_ISGID** bit is cleared.

The access permission bits of *mode* are modified by the process's file mode creation mask: all bits set in the process's file mode creation mask are cleared (see **umask**(2)). If *mode* indicates a block or character special file, *dev* is a configuration-dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored. See **makedev**(3C).

mknod() may be invoked only by a privileged user for file types other than FIFO special.

If *path* is a symbolic link, it is not followed.

RETURN VALUES

0 is returned upon successful completion; otherwise, **–1** is returned and **errno** is set to indicate the error.

ERRORS

mknod() fails and creates no new file if one or more of the following are true:

EEXIST	The named file exists.
EFAULT	*path* points to an illegal address.
EINTR	A signal was caught during the **mknod()** function.
EINVAL	*dev* is invalid.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines and the file system type does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	A component of the path prefix does not exist or is a null pathname.
ENOLINK	*path* points to a remote machine and the link to that machine is no longer active.
ENOSPC	No space is available.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	The effective user of the calling process is not super-user.
EROFS	The directory in which the file is to be created is located on a read-only file system.

SEE ALSO

chmod(2), **exec**(2), **mkdir**(2), **umask**(2), **makedev**(3C), **mkfifo**(3C), **stat**(5)

NOTES

Normally, applications should use the **mkdir**(2) routine to make a directory, since the function **mknod()** may not establish directory entries for **.** (the directory itself) and **..** (the parent directory), and appropriate permissions are not required. Similarly, **mkfifo**(3C) should be used in preference to **mknod()** in order to create FIFOs.

NAME

mmap – map pages of memory

SYNOPSIS

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

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

DESCRIPTION

The function **mmap( )** establishes a mapping between a process's address space and a virtual memory object. The format of the call is as follows:

pa = **mmap(***addr***,** *len***,** *prot***,** *flags***,** *fildes***,** *off***);**

mmap establishes a mapping between the process's address space at an address *pa* for *len* bytes to the memory object represented by the file descriptor *fildes* at offset *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.

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, such as when the file was created just before 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. In other words, **mmap( )** cannot be used to implicitly extend the length of files.

The mapping established by **mmap( )** replaces any previous mappings for the process's pages in the range [*pa, pa + len*).

Mappings established from *fildes* are not removed upon a **close**(2) of that 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:

PROT_READ	Page can be read.
PROT_WRITE	Page can be written.
PROT_EXEC	Page can be executed.
PROT_NONE	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:

MAP_SHARED	Share changes.
MAP_PRIVATE	Changes are private.
MAP_FIXED	Interpret addr exactly.
MAP_NORESERVE	Don't reserve swap space.

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. Either **MAP_SHARED** or **MAP_PRIVATE** must be specified, but not both. The mapping type is retained across a **fork**(2).

Note that the private copy is not created until the first write; until then, other users who have the object mapped **MAP_SHARED** can change the object.

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* 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".

MAP_NORESERVE specifies that no swap space be reserved for a mapping. Without this flag, the creation of a **MAP_PRIVATE** mapping reserves swap space equal to the size of the mapping; when the mapping is written into, the reserved space is employed to hold private copies of the data. A write into a **MAP_NORESERVE** mapping produces results which depend on the current availability of swap space in the system. If space is available, the write succeeds and a private copy of the written page is created; if space is not available, the write fails and a SIGBUS signal is delivered to the writing process. **MAP_NORESERVE** mappings are inherited across **fork**(2); at the time of the **fork**(2) swap space is reserved in the child for all private pages that currently exist in the parent; thereafter the child's mapping behaves as described above.

The parameter *off* is constrained to be aligned and sized according to the value returned by **sysconf()**. 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*).

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 the delivery of a **SIGBUS** signal. **SIGBUS** signals may also be delivered on various file system conditions, including quota exceeded errors.

If the process calls **mlockall**(3C) 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

On success, **mmap()** returns the address at which the mapping was placed (*pa*). On failure it returns **MAP_FAILED** and sets **errno** to indicate an error.

ERRORS

Under the following conditions, **mmap()** fails and sets **errno** to:

EACCES *fildes* is not open for read, regardless of the protection specified, or *fildes* is not open for write and **PROT_WRITE** was specified for a **MAP_SHARED** type mapping.

EAGAIN The mapping could not be locked in memory.

There was insufficient room to reserve swap space for the mapping.

The file to be mapped is already locked using advisory or mandatory record locking. See **fcntl**(2).

EBADF *fildes* is not open.

EINVAL The arguments *addr* (if **MAP_FIXED** was specified) or *off* are not multiples of the page size as returned by **sysconf()**.

The field in *flags* is invalid (neither **MAP_PRIVATE** or **MAP_SHARED**).

The argument *len* has a value less than or equal to **0**.

ENODEV *fildes* refers to an object for which **mmap()** is meaningless, such as a terminal.

ENOMEM **MAP_FIXED** was specified and the range [*addr*, *addr* + *len*) exceeds that allowed for the address space of a process.

MAP_FIXED was "not" specified and there is insufficient room in the address space to effect the mapping.

The composite size of *len* plus the lengths of all previous mmappings exceeds **RLIMIT_VMEM** (see **getrlimit**(2)).

ENXIO The range [*off*, *off* + *len*) is illegal for mmapping to this device.

SEE ALSO

close(2), **fcntl**(2), **fork**(2), **getrlimit**(2), **mprotect**(2), **munmap**(2), **lockf**(3C), **mlockall**(3C), **plock**(3C), **sysconf**(3C)

NOTES

mmap() allows access to resources using address space manipulations instead of the **read**/**write** interface. Once a file is mapped, all a process has to do to access it is use the

data at the address to which the object was mapped. Consider the following pseudo-code:

```
fildes = open(...)
lseek(fildes, offset)
read(fildes, buf, len)
/* use data in buf */
```

Here is a rewrite using **mmap()**:

```
fildes = open(...)
address = mmap((caddr_t) 0, len, (PROT_READ | PROT_WRITE),
        MAP_PRIVATE, fildes, offset)
/* use data at address */
```

NAME

mount – mount a file system

SYNOPSIS

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

int mount(const char *spec, const char *dir, int mflag, /* char *fstype,
        const char *dataptr, int datalen */ ...);
```

DESCRIPTION

mount() requests that a removable file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. *spec* and *dir* are pointers to path names. *fstype* is the file system type, which can be determined by the **sysfs**(2) function. If both the **MS_DATA** and **MS_FSS** flag bits of *mflag* are off, the file system type defaults to the root file system type. Only if either flag is on is *fstype* used to indicate the file system type.

If the **MS_DATA** flag is set in *mflag* the system expects the *dataptr* and *datalen* arguments to be present. Together they describe a block of file-system specific data at address *dataptr* of length *datalen*. This is interpreted by file-system specific code within the operating system and its format depends on the file system type. If a particular file system type does not require this data, *dataptr* and *datalen* should both be zero. Note that **MS_FSS** is obsolete and is ignored if **MS_DATA** is also set, but if **MS_FSS** is set and **MS_DATA** is not, *dataptr* and *datalen* are both assumed to be zero.

After a successful call to **mount()**, all references to the file *dir* refer to the root directory on the mounted file system.

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

The **mount()** system call may only be invoked only by processes with super-user privileges.

RETURN VALUES

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

ERRORS

mount() fails if one or more of the following are true:

EBUSY	*dir* is currently mounted on, is someone's current working directory, or is otherwise busy.
EBUSY	The device associated with *spec* is currently mounted.
EBUSY	There are no more mount table entries.
EFAULT	*spec*, *dir*, or *datalen* points outside the allocated address space of the process.
EINVAL	The super block has an invalid magic number or the *fstype* is invalid.

ELOOP	Too many symbolic links were encountered in translating *spec* or *dir*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines and the file system type does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	None of the named files exists or is a null pathname.
ENOTBLK	*spec* is not a block special device.
ENOTDIR	*dir* is not a directory.
ENOTDIR	A component of a path prefix is not a directory.
EPERM	The effective user ID is not super-user.
EREMOTE	*spec* is remote and cannot be mounted.
ENOLINK	*path* points to a remote machine and the link to that machine is no longer active.
ENXIO	The device associated with *spec* does not exist.
EROFS	*spec* is write protected and *mflag* requests write permission.
ENOSPC	The file system state in the super-block is not **FsOKAY** and *mflag* requests write permission.

SEE ALSO **mount**(1M), **sysfs**(2), **umount**(2)

NAME

mprotect – set protection of memory mapping

SYNOPSIS

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

int mprotect(caddr_t addr, size_t len, int prot);
```

DESCRIPTION

The function **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** and are defined in **<sys/mman.h>** as:

```
PROT_READ       /* page can be read */
PROT_WRITE      /* page can be written */
PROT_EXEC       /* page can be executed */
PROT_NONE       /* page can not be accessed */
```

RETURN VALUES

Upon successful completion, the function **mprotect( )** returns a value of **0**; otherwise, it returns a value of **–1** and sets **errno** to indicate an error.

ERRORS

Under the following conditions, the function **mprotect( )** fails and sets **errno** to:

EACCES *prot* specifies a protection that violates the access permission the process has to the underlying memory object.

EAGAIN the address range [*addr, addr + len*) includes one or more pages that have been locked in memory and that were mapped **MAP_PRIVATE**; *prot* includes **PROT_WRITE**; and the system has insufficient resources to reserve memory for the private pages that may be created. These private pages may be created by store operations into the now-writable address range.

EINVAL *addr* is not a multiple of the page size as returned by **sysconf**.

EINVAL the *len* argument has a value less than or equal to **0**.

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*) may have been changed. If the error occurs on some page at *addr2*, then the protections of all whole pages in the range [*addr, addr2*] will have been modified.

SEE ALSO

mmap(2), **plock**(3C), **mlock**(3C), **mlockall**(3C), **sysconf**(3C)

NAME

msgctl – message control operations

SYNOPSIS

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

int msgctl(int msqid, int 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 access permission 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*. *buf* is ignored.

RETURN VALUES

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

ERRORS

msgctl( ) fails if one or more of the following are true:

EACCES *cmd* is **IPC_STAT** and 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.

EINVAL *cmd* is not a valid command.

EINVAL *cmd* is **IPC_SET** and **msg_perm.uid** or **msg_perm.gid** is not valid.

EPERM *cmd* is **IPC_RMID** or **IPC_SET**. The effective user of the calling process is not super-user, or the value of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*.

EPERM *cmd* is **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 that of super user.

EOVERFLOW *cmd* is **IPC_STAT** and *uid* or *gid* is too large to be stored in the structure pointed to by *buf*.

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_t key, int msgflg);
```

DESCRIPTION

msgget() returns the message queue identifier associated with *key*.

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

key is **IPC_PRIVATE**.

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

On 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 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 to the low-order 9 bits of *msgflg*.

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

msg_ctime is set to the current time.

msg_qbytes is set to the system limit.

RETURN VALUES

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

ERRORS

msgget() fails if one or more of the following are true:

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

EEXIST A message queue identifier exists for *key* but (*msgflg***&IPC_CREAT**) and (*msgflg***&IPC_EXCL**) are both 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), **stdipc**(3C)

NAME

msgop, msgsnd, msgrcv – message operations

SYNOPSIS

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

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

DESCRIPTION

msgsnd( ) sends a message to the queue associated with the message queue identifier specified by *msqid*. *msgp* points to a user defined buffer that must contain first a field of type long integer that will specify the type of the message, and then a data portion that will hold the text of the message. The following is an example of members that might be in a user defined buffer.

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

mtype is a positive integer that can be used by the receiving process for message selection. **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 is not sent and the calling process returns immediately.

If (*msgflg***&IPC_NOWAIT**) is false, the calling process suspends 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 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**(3C).

msgrcv( ) reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the user defined structure pointed to by *msgp*. The structure must contain a message type field followed by the area for the message text (see the structure **mymsg** above). **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 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 returns immediately with a return value of –1 and sets **errno** to **ENOMSG**.

If (*msgflg***&IPC_NOWAIT**) is false, the calling process suspends 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 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**(3C).

RETURN VALUES

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

Upon successful completion, the return value is as follows:

msgsnd( ) returns a value of 0.

msgrcv returns the number of bytes actually placed into *mtext*.

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

ERRORS

msgsnd( ) fails and sends no message 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.
EINVAL	*msqid* is not a valid message queue identifier.
EINVAL	*mtype* is less than 1.
EINVAL	*msgsz* is less than zero or greater than the system-imposed limit.

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 to the process ID of the calling process.

msg_stime is set to the current time.

msgrcv( ) fails and receives no message if one or more of the following are true:

E2BIG	The length of *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.
EINVAL	*msqid* is not a valid message queue identifier.
EINVAL	*msgsz* is less than 0.
ENOMSG	The queue does not contain a message of the desired type and (*msgtyp***&IPC_NOWAIT**) is true.

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 to the process ID of the calling process.

msg_rtime is set to the current time.

SEE ALSO **intro**(2), **msgctl**(2), **msgget**(2), **signal**(3C)

NAME

munmap – unmap pages of memory

SYNOPSIS

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

int munmap(caddr_t addr, size_t len);
```

DESCRIPTION

The function **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.

The function **mmap** often performs an implicit **munmap( )**.

RETURN VALUES

Upon successful completion, the function **munmap( )** returns a value of **0**; otherwise, it returns a value of **–1** and sets **errno** to indicate an error.

ERRORS

Under the following conditions, the function **munmap( )** fails and sets **errno** to:

EINVAL if *addr* is not a multiple of the page size as returned by **sysconf**.

EINVAL if addresses in the range [*addr, addr + len*) are outside the valid range for the address space of a process.

EINVAL The argument *len* has a value less than or equal to **0**.

SEE ALSO

mmap(2), **sysconf**(3C)

NAME

nice – change priority of a process

SYNOPSIS

```
#include <unistd.h>
int nice(int incr);
```

DESCRIPTION

nice( ) allows a process to change its priority. The invoking process must be in a scheduling class that supports the **nice( )** system call. The **priocntl** function is a more general interface to scheduler functions.

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

A maximum nice value of 39 and a minimum nice value of 0 are imposed by the system. (The default nice value is 20.) Requests for values above or below these limits result in the nice value being set to the corresponding limit.

RETURN VALUES

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

ERRORS

nice( ) fails if one or more of the following are true:

EINVAL **nice( )** is called by a process in a scheduling class other than time-sharing.

EPERM *incr* is negative or greater than 40 and the effective user id of the calling process is not super-user.

SEE ALSO

nice(1), **exec**(2), **priocntl**(2)

NAME

open – open for reading or writing

SYNOPSIS

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

int open(const char *path, int oflag, /* mode_t mode */ ...);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

open() opens a file descriptor for the file with the path name pointed to by *path*, and sets the file status flags according to the value of *oflag*. *oflag* values are constructed by OR-ing Flags from the following list (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY or **O_NONBLOCK** These flags may affect subsequent reads and writes (see **read**(2) and **write**(2)). If both **O_NDELAY** and **O_NONBLOCK** are set, **O_NONBLOCK** will take precedence.

When opening a FIFO with **O_RDONLY** or **O_WRONLY** set:

If **O_NDELAY** or **O_NONBLOCK** is set: An **open** for reading-only will return without delay; an **open** for writing-only will return an error if no process currently has the file open for reading.

If **O_NDELAY** and **O_NONBLOCK** are clear: An **open** for reading-only will block until a process opens the file for writing; an **open** for writing-only will block until a process opens the file for reading.

When opening a file associated with a terminal line:

If **O_NDELAY** or **O_NONBLOCK** is set: The open will return without waiting for the device to be ready or available; subsequent behavior of the device is device specific.

If **O_NDELAY** and **O_NONBLOCK** are clear: The open will block until the device is ready or available.

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

O_DSYNC Write I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion.

O_RSYNC Read I/O operations on the file descriptor complete at the same level of integrity as specified by the **O_DSYNC** and **O_SYNC** flags. If both **O_DSYNC** and **O_RSYNC** are set in *oflag*, all I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion. If both **O_SYNC** and **O_RSYNC** are set in *oflag*, all I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.

O_SYNC When opening a regular file, this flag affects subsequent writes. If set, each **write**(2) will wait for both the file data and file status to be physically updated. Write I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.

O_NOCTTY If set and the file is a terminal, the terminal will not be allocated as the calling process's controlling terminal.

O_CREAT If the file exists, this flag has no effect, except as noted under **O_EXCL** below. 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 the effective group ID of the process, or if the S_ISGID bit is set in the directory in which the file is being created, the file's group ID is set to the group ID of its parent directory. If the group ID of the new file does not match the effective group ID or one of the supplementary groups IDs, the **S_ISGID** bit is cleared. The access permission bits of the file mode are set to the value of *mode*, modified as follows (see **creat**(2)):

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

The "save text image after execution bit" of the mode is cleared (see **chmod**(2)). **O_SYNC** Write I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion (see **fcntl**(5) definition of **O_SYNC .)**

O_EXCL If **O_EXCL** and **O_CREAT** are set, **open( )** will fail if the file exists. The check for the existence of the file and the creation of the file if it does not exist is atomic with respect to other processes executing **open( )** naming the same filename in the same directory with **O_EXCL** and **O_CREAT** set.

O_TRUNC If the file exists, its length is truncated to 0 and the mode and owner are unchanged. **O_TRUNC** has no effect on FIFO special files or directories.

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

When **open( )** is invoked to open a named stream, and the **connld** module (see **connld**(7M)) has been pushed on the pipe, **open( )** blocks until the server process has issued an **I_RECVFD ioctl** (see **streamio**(7I)) to receive the file descriptor.

If *path* is a symbolic link and **O_CREAT** and **O_EXCL** are set, the link is not followed.

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

The new file descriptor is the lowest numbered file descriptor available and is set to remain open across **exec** functions (see **fcntl**(2)).

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

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 did previously exist, upon successful completion **open()** marks for update the **st_ctime** and **st_mtime** fields of the file.

RETURN VALUES

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

ERRORS

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

EACCES	The file does not exist and write permission is denied by the parent directory of the file to be created.
	O_TRUNC is specified and write permission is denied
	A component of the path prefix denies search permission.
	oflag permission is denied for an existing file.
EAGAIN	If the file exists with enforced record locking enabled, record locks are on the file (see **chmod**(2)**),** and **O_TRUNC** is specified.
EEXIST	**O_CREAT** and **O_EXCL** are set, and the named file exists.
EFAULT	*path* points to an illegal address.
EINTR	A signal was caught during the **open()** function.
EIO	A hangup or error occurred during the open of the STREAMS-based device.
EISDIR	The named file is a directory and *oflag* is write or read / write.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMFILE	The process has too many open files (see **getrlimit**(2)).
EMULTIHOP	Components of *path* require hopping to multiple remote machines and the file system does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENFILE	The system file table is full.
ENOENT	**O_CREAT** is not set and the named file does not exist.
	O_CREAT is set and a component of the path prefix does not exist or is the null pathname.
ENOLINK	*path* points to a remote machine, and the link to that machine is no

	longer active.
ENOMEM	The system is unable to allocate a send descriptor.
ENOSPC	**O_CREAT** and **O_EXCL** are set, and the file system is out of inodes.
	O_CREAT is set and the directory cannot be extended.
ENOSR	Unable to allocate a stream.
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The named file is a character special or block special file, and the device associated with this special file does not exist.
	O_NDELAY or **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.
EROFS	The named file resides on a read-only file system and either **O_WRONLY , O_RDWR , O_CREAT ,** or **O_TRUNC** is set in *oflag* (if the file does not exist).

SEE ALSO **intro**(2), **chmod**(2), **close**(2), **creat**(2), **dup**(2), **exec**(2), **fcntl**(2), **getmsg**(2), **getrlimit**(2), **lseek**(2), **putmsg**(2), **read**(2), **stat**(2), **umask**(2), **write**(2), **stat**(5), **fcntl**(5), **connld**(7M), **streamio**(7I)

NAME

p_online – change processor online or offline status

SYNOPSIS

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

int p_online(processorid_t processorid , int flag );
```

DESCRIPTION

The processor specified by the first argument is set online or offline or is unchanged, depending on whether the *flag* argument is **P_ONLINE**, **P_OFFLINE**, or **P_STATUS**.

When a flag of **P_ONLINE** is specified, the processor, if previously offline, is brought online and allowed to process LWPs and perform system activities.

When **P_OFFLINE** is specified, and the processor is not already offline, it is taken offline and not allowed to process LWPs. The processor will become as inactive as possible.

When **P_STATUS** is specified, no change occurs, but the current status is returned.

RETURN VALUES

On successful completion, the value returned is the previous state of the processor, **P_ONLINE** or **P_OFFLINE**. Otherwise, a value of –1 is returned and errno is set to indicate the error.

ERRORS

EPERM	The effective user of the calling process is not superuser.
EINVAL	An non-existent processor ID was specified or *flag* was invalid.
EBUSY	*flag* was **P_OFFLINE** and the specified processor is the only online processor, there are currently LWPs bound to the processor, or the processor performs some essential function that cannot be performed by another processor.

SEE ALSO

psradm(1M), **psrinfo**(1M), **processor_bind**(2), **processor_info**(2), **sysconf**(3C)

NAME

pause – suspend process until signal

SYNOPSIS

```
#include <unistd.h>
int pause(void);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

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

If the signal causes termination of the calling process, **pause( )** does not return.

If the signal is caught by the calling process and control is returned from the signal-catching function (see **signal**(3C)), the calling process resumes execution from the point of suspension; with a return value of –1 from **pause( )** and **errno** set to **EINTR**.

SEE ALSO

alarm(2), **kill**(2), **wait**(2), **signal**(3C)

NAME

pipe – create an interprocess channel

SYNOPSIS

#include <unistd.h>

int pipe(int *fildes***[2]);**

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

pipe() creates an I/O mechanism called a pipe and returns two file descriptors, *fildes***[0]** and *fildes***[1]**. The files associated with *fildes***[0]** and *fildes***[1]** are streams and are both opened for reading and writing. The **O_NDELAY** and **O_NONBLOCK** flags are cleared.

A read from *fildes***[0]** accesses the data written to *fildes***[1]** on a first-in-first-out (FIFO) basis and a read from *fildes***[1]** accesses the data written to *fildes***[0]** also on a FIFO basis.

The **FD_CLOEXEC** flag will be clear on both file descriptors.

Upon successful completion **pipe()** marks for update the **st_atime, st_ctime**, and **st_mtime** fields of the pipe.

RETURN VALUES

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

ERRORS

pipe() fails if:

EMFILE If **{OPEN_MAX}-1** or more file descriptors are currently open for this process.

ENFILE A file table entry could not be allocated.

SEE ALSO

sh(1), **fcntl**(2), **getmsg**(2), **poll**(2), **putmsg**(2), **read**(2), **write**(2), **streamio**(7I)

NOTES

Since a pipe is bi-directional, there are two separate flows of data. Therefore, the size (**st_size**) returned by a call to **fstat ()** with argument *fildes***[0]** or *fildes***[1]** is the number of bytes available for reading from *fildes***[0]** or *fildes***[1]** respectively. Previously, the size (**st_size**) returned by a call to **fstat()** with argument *fildes***[1]** (the write-end) was the number of bytes available for reading from *fildes***[0]** (the read-end).

NAME

poll – input/output multiplexing

SYNOPSIS

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

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

DESCRIPTION

poll() provides users with a mechanism for multiplexing input/output over a set of file descriptors that reference open files. **poll()** identifies those files on which a user can send or receive messages, or on which certain events have occurred.

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 */
```

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

POLLIN	Data other than high priority data may be read without blocking. For STREAMS, this flag is set even if the message is of zero length.
POLLRDNORM	Normal data (priority band = 0) may be read without blocking. For STREAMS, this flag is set even if the message is of zero length.
POLLRDBAND	Data from a non-zero priority band may be read without blocking For STREAMS, this flag is set even if the message is of zero length.
POLLPRI	High priority data may be received without blocking. For STREAMS, this flag is set even if the message is of zero length.
POLLOUT	Normal data may be written without blocking.
POLLWRNORM	The same as **POLLOUT**.
POLLWRBAND	Priority data (priority band > 0) may be written. This event only examines bands that have been written to at least once.
POLLERR	An error has occurred on the device or stream. This flag is only valid in the **revents** bitmask; it is not used in the **events** field.
POLLHUP	A hangup has occurred on the stream. This event and **POLLOUT** are mutually exclusive; a stream can never be writable if a hangup has occurred. However, this event and **POLLIN**, **POLLRDNORM**, **POLLRDBAND**, or **POLLPRI** are not mutually exclusive. This flag is only valid in the **revents** bitmask; it is not used in the **events** field.
POLLNVAL	The specified **fd** value does not belong to an open file. 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 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 are 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 **INFTIM** (or –1), **poll()** blocks until a requested event occurs or until the call is interrupted. **poll()** is not affected by the **O_NDELAY** and **O_NONBLOCK** flags.

RETURN VALUES

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

ERRORS

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

EAGAIN	Allocation of internal data structures failed, but the request may be attempted again.
EFAULT	Some argument points to an illegal address.
EINTR	A signal was caught during the **poll()** function.
EINVAL	The argument *nfds* is greater than **{OPEN_MAX}**.

SEE ALSO

intro(2), **getmsg**(2), **getrlimit**(2), **putmsg**(2), **read**(2), **write**(2), **chpoll**(9E)

STREAMS Programming Guide

NOTES

Non-STREAMS drivers use **chpoll**(9E) to implement **poll** on these devises.

NAME

priocntl – process scheduler control

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>

long priocntl(idtype_t idtype, id_t id, int cmd, /* arg */ ...);
```

DESCRIPTION

priocntl( ) provides for control over the scheduling of an active light weight process (**LWP**).

LWPs fall into distinct classes with a separate scheduling policy applied to each class. The two classes currently supported are the real-time class and the time-sharing class. The characteristics of these classes are described under the corresponding headings below. The class attribute of an LWP is inherited across the **fork**(2), **exec**(2) and **_lwp_create**(2) system calls. **priocntl( )** can be used to dynamically change the class and other scheduling parameters associated with a running LWP or set of LWPs given the appropriate permissions as explained below.

In the default configuration, a runnable real-time LWP runs before any other LWP. Therefore, inappropriate use of real-time LWP can have a dramatic negative impact on system performance.

priocntl( ) provides an interface for specifying a process, set of processes or an LWP to which the function is to apply. The **priocntlset**(2) system call provides the same functions as **priocntl( )**, but allows a more general interface for specifying the set of LWPs to which the function is to apply.

For **priocntl( )**, the *idtype* and *id* arguments are used together to specify the set of LWPs. The interpretation of *id* depends on the value of *idtype*. The possible values for *idtype* and corresponding interpretations of *id* are as follows:

P_LWPID *id* is an LWP ID. The **priocntl ( )** system call applies to the LWP with the specified ID within the calling process.

P_PID *id* is a process ID specifying a single process. The **priocntl( )** system call applies to all LWPs currently associated with the specified process.

P_PPID *id* is a parent process ID. The **priocntl( )** system call applies to all LWPs currently associated with processes with the specified parent process ID.

P_PGID *id* is a process group ID. The **priocntl( )** system call applies to all LWPs currently associated with processes in the specified process group.

P_SID *id* is a session ID. The **priocntl( )** system call applies to all LWPs currently associated with processes in the specified session.

P_CID *id* is a class ID (returned by **priocntl( ) PC_GETCID** as explained below). The **priocntl( )** system call applies to all LWPs in the specified class.

P_UID *id* is a user ID. The **priocntl( )** system call applies to all LWPs with this effective user ID.

P_GID *id* is a group ID. The **priocntl()** system call applies to all LWPs with this effective group ID.

P_ALL The **priocntl()** system call applies to all existing LWPs. The value of *id* is ignored. The permission restrictions described below still apply.

An *id* value of **P_MYID** can be used in conjunction with the *idtype* value to specify the calling LWP's LWP ID, parent process ID, process group ID, session ID, class ID, user ID, or group ID.

In order to change the scheduling parameters of an LWP (using the **PC_SETPARMS** command as explained below) the real or effective user ID of the LWP calling **priocntl()** must match the real or effective user ID of the receiving LWP or the effective user ID of the calling LWP must be super-user. These are the minimum permission requirements enforced for all classes. An individual class may impose additional permissions requirements when setting LWPs to that class and/or when setting class-specific scheduling parameters.

A special **sys** scheduling class exists for the purpose of scheduling the execution of certain special system processes (such as the swapper process). It is not possible to change the class of any LWP to **sys**. In addition, any processes in the **sys** class that are included in a specified set of processes are disregarded by **priocntl()**. For example, an *idtype* of **P_UID** and an *id* value of zero would specify all processes with a user ID of zero except processes in the **sys** class and (if changing the parameters using **PC_SETPARMS**) the **init**(1M) process.

The **init** process is a special case. In order for a **priocntl()** call to change the class or other scheduling parameters of the **init** process (process ID 1), it must be the only process specified by *idtype* and *id*. The **init** process may be assigned to any class configured on the system, but the time-sharing class is almost always the appropriate choice. (Other choices may be highly undesirable; see the *System Administration Guide, Volume II* for more information.)

The data type and value of *arg* are specific to the type of command specified by *cmd*.

A structure with the following members is used by the **PC_GETCID** and **PC_GETCLINFO** commands.

```
id_t    pc_cid;                 /* Class id */
char    pc_clname[PC_CLNMSZ];   /* Class name */
long    pc_clinfo[PC_CLINFOSZ]; /* Class information */
```

pc_cid is a class ID returned by **priocntl() PC_GETCID**. **pc_clname** is a buffer of size **PC_CLNMSZ** (defined in **<sys/priocntl.h>**) used to hold the class name (**RT** for real-time or **TS** for time-sharing).

pc_clinfo is a buffer of size **PC_CLINFOSZ** (defined in **<sys/priocntl.h>**) used to return data describing the attributes of a specific class. The format of this data is class-specific and is described under the appropriate heading (REAL-TIME CLASS or TIME-SHARING CLASS) below.

A structure with the following elements is used by the **PC_SETPARMS** and **PC_GETPARMS** commands.

```
id_t    pc_cid;                    /* LWP class */
long    pc_clparms[PC_CLPARMSZ];   /* Class-specific params */
```

pc_cid is a class ID (returned by **priocntl() PC_GETCID**). The special class ID **PC_CLNULL** can also be assigned to **pc_cid** when using the **PC_GETPARMS** command as explained below.

The **pc_clparms** buffer holds class-specific scheduling parameters. The format of this parameter data for a particular class is described under the appropriate heading below. **PC_CLPARMSZ** is the length of the **pc_clparms** buffer and is defined in **<sys/priocntl.h>**.

Commands

Available **priocntl()** commands are:

PC_GETCID

Get class ID and class attributes for a specific class given class name. The *idtype* and *id* arguments are ignored. If *arg* is non-null, it points to a structure of type **pcinfo_t**. The **pc_clname** buffer contains the name of the class whose attributes you are getting.

On success, the class ID is returned in **pc_cid**, the class attributes are returned in the **pc_clinfo** buffer, and the **priocntl()** call returns the total number of classes configured in the system (including the **sys** class). If the class specified by **pc_clname** is invalid or is not currently configured the **priocntl()** call returns –1 with **errno** set to **EINVAL**. The format of the attribute data returned for a given class is defined in the **<sys/rtpriocntl.h>** or **<sys/tspriocntl.h>** header file and described under the appropriate heading below.

If *arg* is a **NULL** pointer, no attribute data is returned but the **priocntl()** call still returns the number of configured classes.

PC_GETCLINFO

Get class name and class attributes for a specific class given class ID. The *idtype* and *id* arguments are ignored. If *arg* is non-null, it points to a structure of type **pcinfo_t**. **pc_cid** is the class ID of the class whose attributes you are getting.

On success, the class name is returned in the **pc_clname** buffer, the class attributes are returned in the **pc_clinfo** buffer, and the **priocntl()** call returns the total number of classes configured in the system (including the **sys** class). The format of the attribute data returned for a given class is defined in the **<sys/rtpriocntl.h>** or **<sys/tspriocntl.h>** header file and described under the appropriate heading below.

If *arg* is a **NULL** pointer, no attribute data is returned but the **priocntl()** call still returns the number of configured classes.

PC_SETPARMS
Set the class and class-specific scheduling parameters of the specified LWP(s) associated with the specified process(es). When this command is used with the *idtype* of P_LWPID, it will set the class and class-specific scheduling parameters of the LWP. *arg* points to a structure of type **pcparms_t**. **pc_cid** specifies the class you are setting and the **pc_clparms** buffer contains the class-specific parameters you are setting. The format of the class-specific parameter data is defined in the **<sys/rtpriocntl.h>** or **<sys/tspriocntl.h>** header and described under the appropriate class heading below.

When setting parameters for a set of LWPs, **priocntl( )** acts on the LWPs in the set in an implementation-specific order. If **priocntl( )** encounters an error for one or more of the target processes, it may or may not continue through the set of LWPs, depending on the nature of the error. If the error is related to permissions (**EPERM**), **priocntl( )** continues through the LWP set, resetting the parameters for all target LWPs for which the calling LWP has appropriate permissions. **priocntl( )** then returns –1 with **errno** set to **EPERM** to indicate that the operation failed for one or more of the target LWPs. If **priocntl( )** encounters an error other than permissions, it does not continue through the set of target LWPs but returns the error immediately.

PC_GETPARMS
Get the class and/or class-specific scheduling parameters of an LWP. *arg* points to a structure of type **pcparms_t**.

If **pc_cid** specifies a configured class and a single LWP belonging to that class is specified by the *idtype* and *id* values or the **procset** structure, then the scheduling parameters of that LWP are returned in the **pc_clparms** buffer. If the LWP specified does not exist or does not belong to the specified class, the **priocntl( )** call returns –1 with **errno** set to **ESRCH**.

If **pc_cid** specifies a configured class and a set of LWPs is specified, the scheduling parameters of one of the specified LWP belonging to the specified class are returned in the **pc_clparms** buffer and the **priocntl( )** call returns the process ID of the selected LWP. The criteria for selecting an LWP to return in this case is class dependent. If none of the specified LWPs exist or none of them belong to the specified class the **priocntl( )** call returns –1 with **errno** set to **ESRCH**.

If **pc_cid** is **PC_CLNULL** and a single LWP is specified the class of the specified LWP is returned in **pc_cid** and its scheduling parameters are returned in the **pc_clparms** buffer.

PC_ADMIN
This command provides functionality needed for the implementation of the **dispadmin**(1M) command. It is not intended for general use by other applications.

REAL-TIME CLASS

The real-time class provides a fixed priority preemptive scheduling policy for those LWPS requiring fast and deterministic response and absolute user/application control of scheduling priorities. If the real-time class is configured in the system it should have exclusive control of the highest range of scheduling priorities on the system. This ensures that a runnable real-time LWP is given CPU service before any LWP belonging to any other class.

The real-time class has a range of real-time priority (**rt_pri**) values that may be assigned to an LWP within the class. Real-time priorities range from 0 to *x*, where the value of *x* is configurable and can be determined for a specific installation by using the **priocntl()** **PC_GETCID** or **PC_GETCLINFO** command.

The real-time scheduling policy is a fixed priority policy. The scheduling priority of a real-time LWP is never changed except as the result of an explicit request by the user/application to change the **rt_pri** value of the LWP.

For an LWP in the real-time class, the **rt_pri** value is, for all practical purposes, equivalent to the scheduling priority of the LWP. The **rt_pri** value completely determines the scheduling priority of a real-time LWP relative to other LWPs within its class. Numerically higher **rt_pri** values represent higher priorities. Since the real-time class controls the highest range of scheduling priorities in the system it is guaranteed that the runnable real-time LWP with the highest **rt_pri** value is always selected to run before any other LWPs in the system.

In addition to providing control over priority, **priocntl()** provides for control over the length of the time quantum allotted to the LWP in the real-time class. The time quantum value specifies the maximum amount of time an LWP may run assuming that it does not complete or enter a resource or event wait state (**sleep**). Note that if another LWP becomes runnable at a higher priority, the currently running LWP may be preempted before receiving its full time quantum.

The system's process scheduler keeps the runnable real-time LWPs on a set of scheduling queues. There is a separate queue for each configured real-time priority and all real-time LWPs with a given **rt_pri** value are kept together on the appropriate queue. The LWPs on a given queue are ordered in FIFO order (that is, the LWP at the front of the queue has been waiting longest for service and receives the CPU first). Real-time LWPs that wake up after sleeping, LWPs which change to the real-time class from some other class, LWPs which have used their full time quantum, and runnable LWPs whose priority is reset by **priocntl()** are all placed at the back of the appropriate queue for their priority. An LWP that is preempted by a higher priority LWP remains at the front of the queue (with whatever time is remaining in its time quantum) and runs before any other LWP at this priority. Following a **fork**(2) or **_lwp_create**(2) system call by a real-time LWP, the parent LWP continues to run while the child LWP (which inherits its parent's **rt_pri** value) is placed at the back of the queue.

A structure with the following members (defined in **<sys/rtpriocntl.h>**) defines the format used for the attribute data for the real-time class.

```
short     rt_maxpri;     /* Maximum real-time priority */
```

The **priocntl() PC_GETCID** and **PC_GETCLINFO** commands return real-time class attributes in the **pc_clinfo** buffer in this format.

rt_maxpri specifies the configured maximum **rt_pri** value for the real-time class (if **rt_maxpri** is *x*, the valid real-time priorities range from 0 to *x*).

A structure with the following members (defined in **<sys/rtpriocntl.h>**) defines the format used to specify the real-time class-specific scheduling parameters of an LWP.

```
short   rt_pri;     /* Real-Time priority */
ulong   rt_tqsecs;  /* Seconds in time quantum */
long    rt_tqnsecs; /* Additional nanoseconds in quantum */
```

When using the **priocntl() PC_SETPARMS** or **PC_GETPARMS** commands, if **pc_cid** specifies the real-time class, the data in the **pc_clparms** buffer is in this format.

The above commands can be used to set the real-time priority to the specified value or get the current **rt_pri** value. Setting the **rt_pri** value of an LWP that is currently running or runnable (not sleeping) causes the LWP to be placed at the back of the scheduling queue for the specified priority. The LWP is placed at the back of the appropriate queue regardless of whether the priority being set is different from the previous **rt_pri** value of the LWP. Note that a running LWP can voluntarily release the CPU and go to the back of the scheduling queue at the same priority by resetting its **rt_pri** value to its current real-time priority value. In order to change the time quantum of an LWP without setting the priority or affecting the LWP's position on the queue, the **rt_pri** field should be set to the special value **RT_NOCHANGE** (defined in **<sys/rtpriocntl.h>**). Specifying **RT_NOCHANGE** when changing the class of an LWP to real-time from some other class results in the real-time priority being set to zero.

For the **priocntl() PC_GETPARMS** command, if **pc_cid** specifies the real-time class and more than one real-time LWP is specified, the scheduling parameters of the real-time LWP with the highest **rt_pri** value among the specified LWPs are returned and the LWP ID of this LWP is returned by the **priocntl()** call. If there is more than one LWP sharing the highest priority, the one returned is implementation-dependent.

The **rt_tqsecs** and **rt_tqnsecs** fields are used for getting or setting the time quantum associated with an LWP or group of LWPs. **rt_tqsecs** is the number of seconds in the time quantum and **rt_tqnsecs** is the number of additional nanoseconds in the quantum. For example setting **rt_tqsecs** to 2 and **rt_tqnsecs** to 500,000,000 (decimal) would result in a time quantum of two and one-half seconds. Specifying a value of 1,000,000,000 or greater in the **rt_tqnsecs** field results in an error return with **errno** set to **EINVAL**. Although the resolution of the **tq_nsecs** field is very fine, the specified time quantum length is rounded up by the system to the next integral multiple of the system clock's resolution. The maximum time quantum that can be specified is implementation-specific and equal to **LONG_MAX** ticks (defined in **<limits.h>**). Requesting a quantum greater than this maximum results in an error return with **errno** set to **ERANGE** (although infinite quantums may be requested using a special value as explained below). Requesting a time quantum of zero (setting both **rt_tqsecs** and **rt_tqnsecs** to 0) results in an error return with **errno** set to **EINVAL**.

The **rt_tqnsecs** field can also be set to one of the following special values (defined in **<sys/rtpriocntl.h>**), in which case the value of **rt_tqsecs** is ignored.

RT_TQINF	Set an infinite time quantum.
RT_TQDEF	Set the time quantum to the default for this priority (see **rt_dptbl**(4)).

RT_NOCHANGE Do nott set the time quantum. This value is useful when you wish to change the real-time priority of an LWP without affecting the time quantum. Specifying this value when changing the class of an LWP to real-time from some other class is equivalent to specifying **RT_TQDEF**.

In order to change the class of an LWP to real-time (from any other class) the LWP invoking **priocntl( )** must have super-user privileges. In order to change the priority or time quantum setting of a real-time LWP, the LWP invoking **priocntl( )** must have super-user privileges or must itself be a real-time LWP whose real or effective user ID matches the real of effective user ID of the target LWP.

The real-time priority and time quantum are inherited across the **fork**(2) and **exec**(2) system calls.

TIME-SHARING CLASS

The time-sharing scheduling policy provides for a fair and effective allocation of the CPU resource among LWPs with varying CPU consumption characteristics. The objectives of the time-sharing policy are to provide good response time to interactive LWPs and good throughput to CPU-bound jobs while providing a degree of user/application control over scheduling.

The time-sharing class has a range of time-sharing user priority (see **ts_upri** below) values that may be assigned to LWPs within the class. A **ts_upri** value of zero is defined as the default base priority for the time-sharing class. User priorities range from $-x$ to $+x$ where the value of x is configurable and can be determined for a specific installation by using the **priocntl( ) PC_GETCID** or **PC_GETCLINFO** command.

The purpose of the user priority is to provide some degree of user/application control over the scheduling of LWPs in the time-sharing class. Raising or lowering the **ts_upri** value of an LWP in the time-sharing class raises or lowers the scheduling priority of the LWP. It is not guaranteed, however, that an LWP with a higher **ts_upri** value will run before one with a lower **ts_upri** value. This is because the **ts_upri** value is just one factor used to determine the scheduling priority of a time-sharing LWP. The system may dynamically adjust the internal scheduling priority of a time-sharing LWP based on other factors such as recent CPU usage.

In addition to the system-wide limits on user priority (returned by the **PC_GETCID** and **PC_GETCLINFO** commands) there is a per LWP user priority limit (see **ts_uprilim** below), which specifies the maximum **ts_upri** value that may be set for a given LWP; by default, **ts_uprilim** is zero.

A structure with the following members (defined in **<sys/tspriocntl.h>**) defines the format used for the attribute data for the time-sharing class.

```
short    ts_maxupri;    /* Limits of user priority range */
```

The **priocntl( ) PC_GETCID** and **PC_GETCLINFO** commands return time-sharing class attributes in the **pc_clinfo** buffer in this format.

ts_maxupri specifies the configured maximum user priority value for the time-sharing class. If **ts_maxupri** is x, the valid range for both user priorities and user priority limits is from $-x$ to $+x$.

A structure with the following members (defined in **<sys/tspriocntl.h>**) defines the format used to specify the time-sharing class-specific scheduling parameters of an LWP.

```
short    ts_uprilim;    /* Time-Sharing user priority limit */
short    ts_upri;       /* Time-Sharing user priority */
```

When using the **priocntl() PC_SETPARMS** or **PC_GETPARMS** commands, if **pc_cid** specifies the time-sharing class, the data in the **pc_clparms** buffer is in this format.

For the **priocntl() PC_GETPARMS** command, if **pc_cid** specifies the time-sharing class and more than one time-sharing LWP is specified, the scheduling parameters of the time-sharing LWP with the highest **ts_upri** value among the specified LWPs is returned and the LWP ID of this LWP is returned by the **priocntl()** call. If there is more than one LWP sharing the highest user priority, the one returned is implementation-dependent.

Any time-sharing LWP may lower its own **ts_uprilim** (or that of another LWP with the same user ID). Only a time-sharing LWP with super-user privileges may raise a **ts_uprilim**. When changing the class of an LWP to time-sharing from some other class, super-user privileges are required in order to set the initial **ts_uprilim** to a value greater than zero. Attempts by a non-super-user LWP to raise a **ts_uprilim** or set an initial **ts_uprilim** greater than zero fail with a return value of –1 and **errno** set to **EPERM**.

Any time-sharing LWP may set its own **ts_upri** (or that of another LWP with the same user ID) to any value less than or equal to the LWP's **ts_uprilim**. Attempts to set the **ts_upri** above the **ts_uprilim** (and / or set the **ts_uprilim** below the **ts_upri**) result in the **ts_upri** being set equal to the **ts_uprilim**.

Either of the **ts_uprilim** or **ts_upri** fields may be set to the special value **TS_NOCHANGE** (defined in **<sys/tspriocntl.h>**) in order to set one of the values without affecting the other. Specifying **TS_NOCHANGE** for the **ts_upri** when the **ts_uprilim** is being set to a value below the current **ts_upri** causes the **ts_upri** to be set equal to the **ts_uprilim** being set. Specifying **TS_NOCHANGE** for a parameter when changing the class of an LWP to time-sharing (from some other class) causes the parameter to be set to a default value. The default value for the **ts_uprilim** is **0** and the default for the **ts_upri** is to set it equal to the **ts_uprilim** which is being set.

The time-sharing user priority and user priority limit are inherited across the **fork** and **exec** functions.

RETURN VALUES

Unless otherwise noted above, **priocntl()** returns a value of 0 on success. **priocntl()** returns –1 on failure and sets **errno** to indicate the error.

ERRORS

priocntl() fails if one or more of the following are true :

EAGAIN An attempt to change the class of an LWP failed because of insufficient resources other than memory (for example, class-specific kernel data structures).

EFAULT One of the arguments points to an illegal address.

EINVAL The argument *cmd* was invalid, an invalid or unconfigured class was specified, or one of the parameters specified was invalid.

ENOMEM	An attempt to change the class of an LWP failed because of insufficient memory.
EPERM	The effective user of the calling LWP is not super-user.
ERANGE	The requested time quantum is out of range.
ESRCH	None of the specified LWPs exist.

SEE ALSO

priocntl(1), **dispadmin**(1M), **init**(1M), **_lwp_create**(2), **exec**(2), **fork**(2), **nice**(2), **priocntlset**(2), **rt_dptbl**(4)

System Interfaces Guide

NAME

priocntlset – generalized process scheduler control

SYNOPSIS

```
#include <sys/types.h>
#include <sys/procset.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>

long priocntlset(procset_t *psp, int cmd, /* arg */ ...);
```

DESCRIPTION

priocntlset( ) changes the scheduling properties of running processes. **priocntlset( )** has the same functions as the **priocntl( )** function, but a more general way of specifying the set of processes whose scheduling properties are to be changed.

cmd specifies the function to be performed. *arg* is a pointer to a structure whose type depends on *cmd*. See **priocntl**(2) for the valid values of *cmd* and the corresponding *arg* structures.

psp is a pointer to a **procset** structure, which **priocntlset( )** uses to specify the set of processes whose scheduling properties are to be changed. The **procset** structure contains the following members:

```
idop_t      p_op;      /* operator connecting left/right sets */
idtype_t    p_lidtype; /* left set ID type */
id_t        p_lid;     /* left set ID */
idtype_t    p_ridtype; /* right set ID type */
id_t        p_rid;     /* right set ID */
```

p_lidtype and **p_lid** specify the ID type and ID of one (“left”) set of processes; **p_ridtype** and **p_rid** specify the ID type and ID of a second (“right”) set of processes. ID types and IDs are specified just as for the **priocntl( )** function. **p_op** specifies the operation to be performed on the two sets of processes to get the set of processes the function is to apply to. The valid values for **p_op** and the processes they specify are:

POP_DIFF	set difference: processes in left set and not in right set
POP_AND	set intersection: processes in both left and right sets
POP_OR	set union: processes in either left or right sets or both
POP_XOR	set exclusive-or: processes in left or right set but not in both

The following macro, which is defined in **procset.h**, offers a convenient way to initialize a **procset** structure:

```
#define setprocset(psp, op, ltype, lid, rtype, rid) \
(psp)→p_op      = (op), \
(psp)→p_lidtype  = (ltype), \
(psp)→p_lid     = (lid), \
(psp)→p_ridtype  = (rtype), \
(psp)→p_rid     = (rid),
```

RETURN VALUES

Unless otherwise noted above, **priocntlset( )** returns a value of 0 on success. **priocntlset( )** returns –1 on failure and sets **errno** to indicate the error.

ERRORS

priocntlset( ) fails if one or more of the following are true :

EAGAIN An attempt to change the class of a process failed because of insufficient resources other than memory (for example, class-specific kernel data structures).

EFAULT One of the arguments points to an illegal address.

EINVAL The argument *cmd* was invalid, an invalid or unconfigured class was specified, or one of the parameters specified was invalid.

ENOMEM An attempt to change the class of a process failed because of insufficient memory.

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

ERANGE The requested time quantum is out of range.

ESRCH None of the specified processes exist.

SEE ALSO

priocntl(1), **priocntl**(2)

NAME

processor_bind – bind LWPs to a processor

SYNOPSIS

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

int processor_bind(idtype_t idtype, id_t id, processorid_t processorid,
    processorid_t *obind);
```

DESCRIPTION

The LWP or set of LWPs specified by *idtype* and *id* are bound to the processor specified by *processorid*. Additionally, if *obind* is not **NULL**, the *processorid_t* variable pointed to by *obind* will be set to the previous binding of one of the specified LWPs, or to **PBIND_NONE** if the selected LWP was not bound.

If *idtype* is **P_PID**, the binding effects all LWPs of the process with process ID (PID) *id*.

If *idtype* is **P_LWPID**, the binding effects the LWP of the current process with LWP ID *id*.

If *id* is **P_MYID**, the specified LWP or process is the current one.

If *processorid* is **PBIND_NONE**, the processor bindings of the specified LWPs are cleared.

If *processorid* is **PBIND_QUERY**, the processor bindings are not changed.

The effective user of the calling process must be super-user, or its real or effective user ID must match the real or effective user ID of the LWPs being bound. If the calling process does not have permission to change all of the specified LWPs, the bindings of the LWPs for which it does have permission will be changed even though an error is returned.

RETURN VALUES

processor_bind returns **0** if successful; otherwise, **–1** is returned and **errno** is set to reflect the error.

ERRORS

ESRCH	No processes or LWPs were found to match the criteria specified by *idtype* and *id*.
EINVAL	An non-existent or offline processor was specified.
EINVAL	*idtype* was not **P_PID** or **P_LWPID**.
EFAULT	The location pointed to by *obind* was not **NULL** and not writable by the user.
EPERM	The effective user of the calling process is not super-user, and its real or effective user ID does not match the real or effective user ID of one of the LWPs being bound.

SEE ALSO

psradm(1M), **psrinfo**(1M), **p_online**(2), **sysconf**(3C)

NAME

processor_info – determine type and status of a processor

SYNOPSIS

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

int processor_info(processorid_t processorid, processor_info_t *infop);
```

DESCRIPTION

The status of the processor specified by *processorid* is returned in the **processor_info_t** structure pointed to by *infop*.

The structure contains the following members:

```
int    pi_state;                    /* P_ONLINE or P_OFFLINE*/
char   pi_processor_type[PI_TYPELEN];
char   pi_fputypes[PI_FPUTYPE];
int    pi_clock;                    /* CPU clock freq in MHz */
```

The fields have the following meanings:

pi_state is the current state of the processor, either **P_ONLINE** or **P_OFFLINE**.

pi_processor_type is a NULL-terminated ASCII string specifying the type of the processor.

pi_fputypes is a NULL-terminated ASCII string containing the comma-separated types of floating-point units (FPUs) attached to the processor. This string will be empty if no FPU is attached.

pi_clock is the processor clock frequency rounded to the nearest megahertz. It may be 0 if not known.

RETURN VALUES

processor_info returns 0 if successful. Otherwise -1 is returned and **errno** is set to reflect the error.

ERRORS

EINVAL	An non-existent processor ID was specified.
EFAULT	The **processor_info_t** structure pointed to by *infop* was not writable by the user.

SEE ALSO

psradm(1M), **psrinfo**(1M), **p_online**(2), **sysconf**(3C)

NAME

profil – execution time profile

SYNOPSIS

```
#include <unistd.h>
void profil(unsigned short *buff, unsigned int bufsiz, unsigned int offset,
        unsigned int scale);
```

DESCRIPTION

profil() provides CPU-use statistics by profiling the amount of CPU time expended by a program. **profil()** generates the statistics by creating an execution histogram for a current process. The histogram is defined for a specific region of program code to be profiled, and the identified region is logically broken up into a set of equal size subdivisions, each of which corresponds to a count in the histogram. With each clock tick, the current subdivision is identified and its corresponding histogram count is incremented. These counts establish a relative measure of how much time is being spent in each code subdivision. The resulting histogram counts for a profiled region can be used to identify those functions that consume a disproportionately high percentage of CPU time.

buff is a buffer of *bufsiz* bytes in which the histogram counts are stored in an array of **unsigned short int**.

offset, *scale*, and *bufsiz* specify the region to be profiled.

offset is effectively the start address of the region to be profiled.

scale, broadly speaking, is a contraction factor that indicates how much smaller the histogram buffer is than the region to be profiled. More precisely, *scale* is interpreted as an unsigned 16-bit fixed-point fraction with the decimal point implied on the left. Its value is the reciprocal of the number of bytes in a subdivision, per byte of histogram buffer. Since there are two bytes per histogram counter, the effective ratio of subdivision bytes per counter is one half the scale.

Several observations can be made:

- the maximal value of *scale*, **0xffff** (approximately 1), maps subdivisions 2 bytes long to each counter.
- the minimum value of *scale* (for which profiling is performed), **0x0002** (1/32,768), maps subdivision 65,536 bytes long to each counter.
- the default value of *scale* (currently used by **cc –qp**), **0x4000**, maps subdivisions 8 bytes long to each counter.

The values are used within the kernel as follows: when the process is interrupted for a clock tick, the value of *offset* is subtracted from the current value of the program counter (pc), and the remainder is multiplied by *scale* to derive a result. That result is used as an index into the histogram array to locate the cell to be incremented. Therefore, the cell count represents the number of times that the process was executing code in the subdivision associated with that cell when the process was interrupted.

scale can be computed as (*RATIO* * **0200000L**), where *RATIO* is the desired ratio of *bufsiz* to profiled region size, and has a value between 0 and 1. Qualitatively speaking, the closer *RATIO* is to 1, the higher the resolution of the profile information.

bufsiz can be computed as (*size_of_region_to_be_profiled* * *RATIO*).

SEE ALSO

exec(2), **fork**(2), **times**(2), **monitor**(3C), **prof**(5)

NOTES

Profiling is turned off by giving a *scale* of 0 or 1, and is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an **exec**(2) is executed, but remains on in both child and parent processes after a **fork**(2). Profiling is turned off if a *buff* update would cause a memory fault.

NAME

ptrace – allows a parent process to control the execution of a child process

SYNOPSIS

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

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

DESCRIPTION

ptrace( ) allows a parent process to control the execution of a child process. Its primary use is for the implementation of breakpoint debugging. The child process behaves normally until it encounters a signal (see **signal**(5)), at which time it enters a stopped state and its parent is notified via the **wait**(2) function. When the child is in the stopped state, its parent can examine and modify its "core image" using **ptrace( )**. Also, the parent can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

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

0 This request must be issued by the child process if it is to be traced by its parent. It turns on the child's trace flag that stipulates that the child should be left in a stopped state on receipt of a signal rather than the state specified by *func* (see **signal**(3C)). The *pid*, *addr*, and *data* arguments are ignored, and a return value is not defined for this request. Peculiar results ensue if the parent does not expect to trace the child.

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

1, 2 With these requests, the word at location *addr* in the address space of the child is returned to the parent process. If instruction and data space are separated, request **1** returns a word from instruction space, and request **2** returns a word from data space. If instruction and data space are not separated, either request **1** or request **2** may be used with equal results. The *data* argument is ignored. These two requests fail if *addr* is not the start address of a word, in which case a value of –1 is returned to the parent process and the parent's **errno** is set to **EIO.**

3 With this request, the word at location *addr* in the child's user area in the system's address space (see **<sys/user.h>**) is returned to the parent process. The *data* argument is ignored. This request fails if *addr* is not the start address of a word or is outside the user area, in which case a value of –1 is returned to the parent process and the parent's **errno** is set to **EIO.**

4, 5 With these requests, the value given by the *data* argument is written into the address space of the child at location *addr*. If instruction and data space are separated, request **4** writes a word into instruction space, and request **5** writes a word into data space. If instruction and data space are not separated, either request **4** or request **5** may be used with equal results. On success, the value written into the address space of the child is returned to the parent. These two

requests fail if *addr* is not the start address of a word. On failure a value of –1 is returned to the parent process and the parent's **errno** is set to **EIO.**

6 With this request, a few entries in the child's user area can be written. *data* gives the value that is to be written and *addr* is the location of the entry. The few entries that can be written are the general registers and the condition codes of the Processor Status Word.

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

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

9 This request sets the trace bit in the Processor Status Word of the child and then executes the same steps as listed above for request **7**. The trace bit causes an interrupt on completion of one machine instruction. This effectively allows single stepping of the child.

To forestall possible fraud, **ptrace( )** inhibits the set-user-ID facility on subsequent **exec**(2) calls. If a traced process calls **exec**(2), it stops before executing the first instruction of the new image showing signal **SIGTRAP**.

ERRORS

ptrace( ) in general fails if one or more of the following are true:

EIO *request is an illegal number.*

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

ESRCH *pid* identifies a child that does not exist or has not executed a **ptrace( )** with request **0**.

SEE ALSO

exec(2), **exit**(2), **wait**(2), **signal**(3C), **signal**(5)

NAME

putmsg, putpmsg – send a message on a stream

SYNOPSIS

```
#include <stropts.h>

int putmsg(int fildes, const struct strbuf *ctlptr, const struct strbuf *dataptr, int flags);

int putpmsg(int fildes, const struct strbuf *ctlptr, const struct strbuf *dataptr, int band,
        int flags);
```

DESCRIPTION

putmsg() creates a message 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.

The function **putpmsg()** does the same thing as **putmsg()**, but provides the user the ability to send messages in different priority bands. Except where noted, all information pertaining to **putmsg()** also pertains to **putpmsg()**.

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

```
int maxlen;     /* not used here */
int len;        /* length of data */
void *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* indicates what type of message should be sent and is described later.

To send the data part of a message, *dataptr* must not be **NULL** and the **len** field of *dataptr* must have a value of 0 or greater. To send the control part of a message, the corresponding values must be set for *ctlptr*. No data (control) part is sent if either *dataptr* (*ctlptr*) is **NULL** or the **len** field of *dataptr* (*ctlptr*) is negative.

For **putmsg()**, if a control part is specified, and *flags* is set to **RS_HIPRI**, a high 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 *flags* is set to 0, a normal (non-priority) message is sent. If no control part and no data part are specified, and *flags* is set to 0, no message is sent, and 0 is returned.

The stream head guarantees that the control part of a message generated by **putmsg()** is at least 64 bytes in length.

For **putpmsg()**, the flags are different. *flags* is a bitmask with the following mutually-exclusive flags defined: **MSG_HIPRI** and **MSG_BAND**. If *flags* is set to 0, **putpmsg()** fails and sets **errno** to **EINVAL.** If a control part is specified and *flags* is set to **MSG_HIPRI** and *band* is set to 0, a high-priority message is sent. If *flags* is set to **MSG_HIPRI** and either no control part is specified or *band* is set to a non-zero value, **putpmsg()** fails and sets **errno** to **EINVAL.** If flags is set to **MSG_BAND**, then a message is sent in the priority band specified by *band*. If a control part and data part are not specified and *flags* is set to **MSG_BAND**, no message is sent and 0 is returned.

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

putmsg() or **putpmsg()** 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** or **O_NONBLOCK** has been specified. No partial message is sent.

RETURN VALUES

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

ERRORS

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

EAGAIN	A non-priority message was specified, the **O_NDELAY** or **O_NONBLOCK** flag is set and the stream write queue is full due to internal flow control conditions.
EBADF	*fildes* is not a valid file descriptor open for writing.
EFAULT	*ctlptr* or *dataptr* points to an illegal address.
EINTR	A signal was caught during the **putmsg()** function.
EINVAL	An undefined value was specified in *flags*, or *flags* is set to **RS_HIPRI** and no control part was supplied.
EINVAL	The stream referenced by *fildes* is linked below a multiplexor.
EINVAL	For **putpmsg()**, if *flags* is set to **MSG_HIPRI** and *band* is nonzero.
ENOSR	Buffers could not be allocated for the message that was to be created due to insufficient STREAMS memory resources.
ENOSTR	*fildes* is not associated with a stream.
ENXIO	A hangup condition was generated downstream for the specified stream, or the other end of the pipe is closed.
ERANGE	The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost stream module. This value is also returned if the control part of the message is larger than the maximum configured size of the control part of a message, or if the data part of a message is larger than the maximum configured size of the data part of a message.

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

intro(2), **getmsg**(2), **poll**(2), **read**(2), **write**(2)

STREAMS Programming Guide

NAME

read, pread, readv – read from file

SYNOPSIS

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

ssize_t read(int fildes, void *buf, size_t nbyte);

ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);

ssize_t readv(int fildes, struct iovec *iov, int iovcnt);
```

MT-LEVEL

read() is Async-Signal-Safe

DESCRIPTION

read() attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*. If *nbyte* is zero, **read()** returns zero and has no other results. *fildes* is an open file descriptor.

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

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

pread() performs the same action as **read()**, except that it reads from a given position in the file without changing the file pointer. The first three arguments to **pread()** are the same as **read()** with the addition of a fourth argument *offset* for the desired position inside the file. An attempt to perform a **pread()** on a file that is incapable of seeking results in an error.

readv() performs the same action as **read()**, but places the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], ..., *iov[iovcnt*–1].

The **iovec** structure contains the following members:

```
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()** always fills one buffer completely before proceeding to the next.

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

read() reads data previously written to a file. If any portion of an ordinary file prior to the end of file has not been written, **read()** returns the number of bytes read as **0**. For example, the **lseek** routine allows the file pointer to be set beyond the end of existing data in the file. If additional data is written at this point, subsequent reads in the gap between

the previous end of data and newly written data return bytes with a value of **0** until data is written into the gap.

A **read( )** or **readv( )** from a STREAMS (see **intro**(2)) file can operate in three different modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default is byte-stream mode. This can be changed using the **I_SRDOPT ioctl**(2) request (see **streamio**(7I)), and can be tested with the **I_GRDOPT ioctl**(2) request.

In byte-stream mode, **read( )** and **readv( )** retrieve data from the stream until they have retrieved *nbyte* bytes, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

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

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

If **O_NDELAY** or **O_NONBLOCK** is set, **read( )** returns –1 and sets **errno** to **EAGAIN**.

If **O_NDELAY** and **O_NONBLOCK** are clear, **read( )** sleeps until the blocking record lock is removed.

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

If no process has the pipe open for writing, **read( )** returns **0** to indicate end-of-file.

If some process has the pipe open for writing and **O_NDELAY** is set, **read( )** returns **0.**

If some process has the pipe open for writing and **O_NONBLOCK** is set, **read( )** returns –1 and sets **errno** to **EAGAIN**.

If **O_NDELAY** and **O_NONBLOCK** are clear, **read( )** blocks until data is written to the pipe or the pipe is closed by all processes that had opened the pipe for writing.

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

If **O_NDELAY** is set, **read( )** returns **0.**

If **O_NONBLOCK** is set, **read( )** returns –1 and sets **errno** to **EAGAIN**.

If **O_NDELAY** and **O_NONBLOCK** are clear, **read( )** blocks until data become available.

When attempting to read a file associated with a stream that is not a pipe or FIFO, or terminal, and that has no data currently available:

If **O_NDELAY** or **O_NONBLOCK** is set, **read( )** returns –1 and sets **errno** to **EAGAIN**.

If **O_NDELAY** and **O_NONBLOCK** are clear, **read( )** blocks until data becomes available.

When reading from a STREAMS file, handling of zero-byte messages is determined by the current read mode setting. In byte-stream mode, **read( )** accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. **read( )** then returns the number of bytes read, and places the zero-byte message back on the stream to be retrieved by the next **read( )** or **getmsg( )** (see **getmsg**(2)). 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 returns the data in the message at the front of the stream head read queue, regardless of the priority band of the message.

Normally, a **read( )** from a STREAMS file can only process messages with data and without control information. The **read( )** fails if a message containing control information is encountered at the stream head. This default action can be changed by placing the stream in either control-data mode or control-discard mode with the **I_SRDOPT ioctl**(2). In control-data mode, control messages are converted to data messages by **read( )**. In control-discard mode, control messages are discarded by **read( )**, but any data associated with the control messages is returned to the user.

RETURN VALUES

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

ERRORS

read( ), **pread( )**, and **readv( )** fail if one or more of the following are true:

EAGAIN	Mandatory file / record locking was set, **O_NDELAY** or **O_NONBLOCK** was set, and there was a blocking record lock.
EAGAIN	Total amount of system memory available when reading using raw I / O is temporarily insufficient.
EAGAIN	No data is waiting to be read on a file associated with a tty device and **O_NONBLOCK** was set.
EAGAIN	No message is waiting to be read on a stream and **O_NDELAY** or **O_NONBLOCK** was set.
EBADF	*fildes* is not a valid file descriptor open for reading.
EBADMSG	Message waiting to be read on a stream is not a data message.
EDEADLK	The read was going to go to sleep and cause a deadlock to occur.
EFAULT	*buf* points to an illegal address.

EINTR A signal was caught during the read operation and no data was transferred.

EINVAL Attempted to read from a stream linked to a multiplexor.

EIO A physical I / O error has occurred, or the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group of the process is orphaned.

EISDIR *fildes* refers to a directory on a file system type that does not support read operations on directories.

ENOLCK The system record lock table was full, so the **read( )** or **readv( )** could not go to sleep until the blocking record lock was removed.

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

ENXIO The device associated with *fildes* is a block special or character special file and the value of the file pointer is out of range.

In addition, **readv( )** may return one of the following errors:

EFAULT *iov* points outside the allocated address space.

EINVAL *iovcnt* was less than or equal to **0,** or greater than or equal to **{IOV_MAX}**. (See **intro**(2) for a definition of **{IOV_MAX}**).

EINVAL The sum of the **iov_len** values in the *iov* array overflowed an int.

In addition, **pread( )** fails and the file pointer remains unchanged if the following is true:

ESPIPE *fildes* is associated with a pipe or fifo.

A **read( )** from a STREAMS file also fails 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( )** continues to operate normally until the stream head read queue is empty. Thereafter, it returns **0.**

SEE ALSO

intro(2), **chmod**(2), **creat**(2), **dup**(2), **fcntl**(2), **getmsg**(2), **ioctl**(2), **open**(2), **pipe**(2), **streamio**(7I), **termio**(7I)

NAME

readlink – read the value of a symbolic link

SYNOPSIS

#include <unistd.h>

int readlink(const char **path*, **void** **buf*, **size_t** *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

Upon successful completion **readlink()** returns the number of characters placed in the buffer; otherwise, it returns –1 and places an error code in **errno**.

ERRORS

readlink() fails and the buffer remains unchanged if:

EACCES	Search permission is denied for a component of the path prefix of *path*.
EFAULT	*path* or *buf* points to an illegal address.
EINVAL	The named file is not a symbolic link.
EIO	An I/O error occurs while reading from or writing to the file system.
ELOOP	Too many symbolic links are encountered in translating *path*.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	The named file does not exist.
ENOSYS	The file system does not support symbolic links.

SEE ALSO

stat(2), **symlink**(2)

NAME

rename – change the name of a file

SYNOPSIS

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

The function **rename()** changes the name of a file. *old* points to the pathname of the file to be renamed. *new* points to the new pathname of the file.

If *old* and *new* both refer to the same existing file, the **rename()** function returns successfully and performs no other action.

If *old* points to the pathname of a file that is not a directory, *new* must not point to the pathname of a directory. If the link named by *new* exists, it will be removed and *old* will be renamed to *new*. In this case, a link named *new* must remain visible to other processes throughout the renaming operation and will refer to either the file referred to by *new* or the file referred to as *old* before the operation began.

If *old* points to the pathname of a directory, *new* must not point to the pathname of a file that is not a directory. If the directory named by *new* exists, it will be removed and *old* will be renamed to *new*. In this case, a link named *new* will exist throughout the renaming operation and will refer to either the file referred to by *new* or the file referred to as *old* before the operation began. Thus, if *new* names an existing directory, it must be an empty directory.

The *new* pathname must not contain a path prefix that names *old*. Write access permission is required for both the directory containing *old* and the directory containing *new*. If *old* points to the pathname of a directory, write access permission is required for the directory named by *old*, and, if it exists, the directory named by *new*.

If the directory containing *old* has the sticky bit set, at least one of the following conditions listed below must be true:

- the user must own *old*
- the user must own the directory containing *old*
- *old* must be writable by the user
- the user must be a privileged user

If *new* exists, and the directory containing *new* is writable and has the sticky bit set, at least one of the following conditions must be true:

- the user must own *new*
- the user must own the directory containing *new*
- *new* must be writable by the user
- the user must be a privileged user

If the link named by *new* exists, the file's link count becomes zero when it is removed, and no process has the file open, then the space occupied by the file will be freed and the file will no longer be accessible. If one or more processes have the file open when the last link is removed, the link will be removed before **rename()** returns, but the removal of the file contents will be postponed until all references to the file have been closed.

Upon successful completion, the **rename()** function will mark for update the **st_ctime** and **st_mtime** fields of the parent directory of each file.

RETURN VALUES

Upon successful completion, the function **rename()** returns a value of **0**; otherwise, it returns a value of **−1** and sets **errno** to indicate an error.

ERRORS

Under the following conditions, the function **rename()** fails, and sets **errno** to:

EACCES	A component of either path prefix denies search permission; one of the directories containing *old* and *new* denies write permissions; or write permission is denied by a directory pointed to by *old* or *new*.
EBUSY	*new* is a directory and the mount point for a mounted file system.
EEXIST	The link named by *new* is a directory containing entries other than '**.**' (the directory itself) and '**..**' (the parent directory).
EINVAL	*new* directory pathname contains a path prefix that names the *old* directory.
EISDIR	*new* points to a directory but *old* points to a file that is not a directory.
ELOOP	Too many symbolic links were encountered in translating the pathname.
ENAMETOOLONG	The length of *old* or *new* exceeds **{PATH_MAX},** or a pathname component is longer than **{NAME_MAX}** while **{_POSIX_NO_TRUNC}** is in effect.
EMLINK	The file named by *old* is a directory, and the link count of the parent directory of *new* would exceed **{LINK_MAX}.**
ENOENT	The link named by *old* does not exist, or either *old* or *new* points to an empty string.
ENOSPC	The directory that would contain *new* cannot be extended.
ENOTDIR	A component of either path prefix is not a directory, or *old* names a directory and *new* names a nondirectory file.
EROFS	The requested operation requires writing in a directory on a read-only file system.
EXDEV	The links named by *old* and *new* are on different file systems.
EIO	An I/O error occurred while making or updating a directory entry.

SEE ALSO **chmod**(2), **link**(2), **unlink**(2)

NOTES The system can deadlock if there is a loop in the file system graph. Such a loop takes the form of an entry in directory *a*, say *a/name1*, being a hard link to directory *b*, and an entry in directory *b*, say *b/name2*, being a hard link to directory *a*. When such a loop exists and two separate processes attempt to rename *a/name1* to *b/name2* and rename *b/name2* to *a/name1*, respectively, the system may deadlock attempting to lock both directories for modification. Use symbolic links instead of hard links for directories.

NAME

rmdir – remove a directory

SYNOPSIS

#include <unistd.h>

int rmdir(const char **path*);

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

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

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

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

ERRORS

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

EACCES	Search permission is denied for a component of the path prefix.
EACCES	Write permission is denied on the directory containing the directory to be removed.
EACCES	The parent directory has the **S_ISVTX** variable set and is not owned by the user; the directory is not owned by the user and is not writable by the user; the user is not a super-user.
EBUSY	The directory to be removed is the mount point for a mounted file system.
EEXIST	The directory contains entries other than those for "**.**" and "**..**".
EFAULT	*path* points to an illegal address.
EINVAL	The directory to be removed is the current directory.
EINVAL	The final component of *path* is ''.''.
EIO	An I/O error occurred while accessing the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines and the file system does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.

ENOENT	The named directory does not exist or is the null pathname.
ENOLINK	*path* points to a remote machine, and the connection to that machine is no longer active.
ENOTDIR	A component of the path prefix is not a directory.
EROFS	The directory entry to be removed is part of a read-only file system.

SEE ALSO **mkdir**(1), **rm**(1), **mkdir**(2)

NAME

semctl – semaphore control operations

SYNOPSIS

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

int semctl(int semid, int semnum, int cmd, ...);
```

DESCRIPTION

semctl( ) provides a variety of semaphore control operations as specified by *cmd*. The fourth argument is optional, depending upon the operation requested. If required it is of type **union semun**, which must be explicitly declared by the application program.

```
union semun {
        int val;
        struct semid_ds *buf;
        ushort *array
} arg;
```

The permission required for a semaphore operation is given as {*token*}, where *token* is the type of permission needed. The types of permission are interpreted as follows:

00400	READ by user
00200	ALTER by user
00040	READ by group
00020	ALTER by group
00004	READ by others
00002	ALTER by others

See the **Semaphore Operation Permissions** subsection of the **DEFINITIONS** section of **intro**(2) for more information. The following semaphore operations as specified by *cmd* 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 command is successfully executed, the **semadj** value corresponding to the specified semaphore in all processes is cleared.

GETPID Return the value of **(int) sempid**. {READ}

GETNCNT Return the value of **semncnt**. {READ}

GETZCNT Return the value of **semzcnt**. {READ}

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

GETALL Place **semval**s into array pointed to by *arg.***array**. {READ}

SETALL Set **semval**s 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 operations 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 access permission bits */
```

This command can be executed only 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 command 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*.

RETURN VALUES

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

GETVAL the value of **semval**

GETPID the value of **(int) sempid**

GETNCNT the value of **semncnt**

GETZCNT the value of **semzcnt**

All other successful completions return **0**; otherwise, **–1** is returned and **errno** is set to indicate the error.

ERRORS

semctl( ) fails if one or more of the following are true:

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

EINVAL *semid* is not a valid semaphore identifier.

EINVAL *semnum* is less than **0** or greater than **sem_nsems –1**.

EINVAL *cmd* is not a valid command.

EINVAL *cmd* is **IPC_SET** and **sem_perm.uid** or **sem_perm.gid** is not valid.

EPERM *cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user of the calling process is not super-user, or to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid*.

EOVERFLOW *cmd* is **IPC_STAT** and *uid* or *gid* is too large to be stored in the structure pointed to by *arg.buf*.

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 **ipcs**(1), **intro**(2), **semget**(2), **semop**(2)

NAME

semget – get set of semaphores

SYNOPSIS

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

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

DESCRIPTION

semget( ) returns the semaphore identifier associated with *key*.

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

- *key* is equal to **IPC_PRIVATE**.
- *key* does not already have a semaphore identifier associated with it, and (*semflg***&IPC_CREAT**) is true.

On 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 access permission bits of **sem_perm.mode** are set equal to the access permission 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.

RETURN VALUES

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

ERRORS

semget( ) fails if one or more of the following are true:

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 both (*semflg***&IPC_CREAT**) and (*semflg***&IPC_EXCL**) are both true.
EINVAL	*nsems* is either less than or equal to zero or greater than the system-imposed limit.
EINVAL	A semaphore identifier exists for *key*, but the number of semaphores in the set associated with it is less than *nsems*, and *nsems* is not equal to zero.
ENOENT	A semaphore identifier does not exist for *key* and (*semflg***&IPC_CREAT**) is false.

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

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

SEE ALSO

ipcs(1), **ipcrm**(1), **intro**(2), **semctl**(2), **semop**(2), **stdipc**(3C)

NAME

semop – semaphore operations

SYNOPSIS

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

int semop(int semid, struct sembuf *sops, size_t 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*. The permission required for a semaphore operation is given as {*token*}, where *token* is the type of permission needed. The types of permission are interpreted as follows:

```
00400   READ by user
00200   ALTER by user
00040   READ by group
00020   ALTER by group
00004   READ by others
00002   ALTER by others
```

See the *Semaphore Operation Permissions* section of **intro**(2) for more information.

sem_op specifies the {ALTER} token if its value is negative or positive, and the {READ} token if its value is zero. Depending on the value of *sem_op*, the following may occur:

sem_op **is a negative integer;** {ALTER}

- If **semval** (see **intro**(2)) is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from **semval**. Also, if (*sem_flg***&SEM_UNDO**) is true, the absolute value of *sem_op* is added to the calling process's **semadj** value (see **exit**(2)) for the specified semaphore.
- If **semval** is less than the absolute value of *sem_op* and (*sem_flg***&IPC_NOWAIT**) is true, **semop( )** returns immediately.
- If **semval** is less than the absolute value of *sem_op* and (*sem_flg***&IPC_NOWAIT**) is false, **semop( )** increments the **semncnt** associated with the specified semaphore and suspends 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**(3C).

sem_op **is a positive integer;** {ALTER}

- 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.

sem_op **is zero;** {READ}

- If **semval** is zero, **semop( )** returns immediately.
- If **semval** is not equal to zero and (*sem_flg***&IPC_NOWAIT**) is true, **semop( )** returns immediately.
- If **semval** is not equal to zero and (*sem_flg***&IPC_NOWAIT**) is false, **semop( )** increments the **semzcnt** associated with the specified semaphore and suspends 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**(3C).

RETURN VALUES

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

ERRORS

semop( ) fails if one or more of the following are true for any of the semaphore operations specified by *sops*:

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	**semop( )** A *semid* was removed from the system.
EINTR	A signal was received.
EINVAL	*semid* is not a valid semaphore identifier, or 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 a **semadj** value to overflow the system-imposed limit.

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

SEE ALSO **ipcs**(1), **intro**(2), **exec**(2), **exit**(2), **fork**(2), **semctl**(2), **semget**(2)

NAME

setpgid – set process group ID

SYNOPSIS

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

int setpgid(pid_t pid, pid_t pgid);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

setpgid( ) sets the process group ID of the process with ID *pid* to *pgid*. If *pgid* is equal to *pid*, the process becomes a process group leader. See **intro**(2) for more information on session leaders and process group leaders. If *pgid* is not equal to *pid*, the process becomes a member of an existing process group.

If *pid* is equal to 0, the process ID of the calling process is used. If *pgid* is equal to 0, the process specified by *pid* becomes a process group leader.

RETURN VALUES

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

ERRORS

setpgid( ) fails and returns an error are true:

EACCES *pid* matches the process ID of a child process of the calling process and the child process has successfully executed an **exec**(2) function.

EINVAL *pgid* is less than (**pid_t) 0**, or greater than or equal to {**PID_MAX**}.

EINVAL The calling process has a controlling terminal that does not support job control.

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

EPERM *pid* 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.

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

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

SEE ALSO

intro(2), **exec**(2), **exit**(2), **fork**(2), **getpid**(2), **getsid**(2)

NAME

setpgrp – set process group ID

SYNOPSIS

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

pid_t setpgrp(void);
```

DESCRIPTION

If the calling process is not already a session leader, **setpgrp( )** makes it one by setting its process group ID and session ID to the value of its process ID, and releases its controlling terminal. See **intro**(2) for more information on process group IDs and session leaders.

RETURN VALUES

setpgrp( ) returns the value of the new process group ID.

SEE ALSO

intro(2), **exec**(2), **fork**(2), **getpid**(2), **getsid**(2), **kill**(2), **signal**(3C)

NOTES

setpgrp( ) will be phased out in favor of the **setsid ( )** function.

NAME

setregid – set real and effective group IDs

SYNOPSIS

```
#include <unistd.h>
int setregid(gid_t rgid, gid_t 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**(2), 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 **<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(2), **getgid**(2), **setreuid**(2), **setuid**(2),

NAME

setreuid – set real and effective user IDs

SYNOPSIS

```
#include <unistd.h>
int setreuid(uid_t ruid, uid_t 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**(2) 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 **<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(2), **getuid**(2), **setregid**(2), **setuid**(2)

NAME

setuid, setegid, seteuid, setgid – set user and group IDs

SYNOPSIS

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

int setuid(uid_t uid);

int setegid(gid_t egid);

int seteuid(uid_t euid);

int setgid(gid_t gid);
```

MT-LEVEL

setuid( )and**setgid( )** are Async-Signal-Safe

DESCRIPTION

The **setuid( )** function sets the real user ID, effective user ID, and saved user ID of the calling process. The **setgid( )** function sets the real group ID, effective group ID, and saved group ID of the calling process. The **setegid( )** and **seteuid( )** functions set the effective group and user ID's respectively for the calling process. See **intro**(2) for more information on real, effective, and saved user and group IDs.

At login time, the real user ID, effective user ID, and saved user ID of the login process are set to the login ID of the user responsible for the creation of the process. The same is true for the real, effective, and saved group IDs; they are set to the group ID of the user responsible for the creation of the process.

When a process calls **exec**(2) to execute a file (program), the user and/or group identifiers associated with the process can change. If the file executed is a set-user-ID file, the effective and saved user IDs of the process are set to the owner of the file executed. If the file executed is a set-group-ID file, the effective and saved group IDs of the process are set to the group of the file executed. If the file executed is not a set-user-ID or set-group-ID file, the effective user ID, saved user ID, effective group ID, and saved group ID are not changed.

The following subsections describe the behavior of **setuid( )** and **setgid( )** with respect to the three types of user and group IDs.

If the effective user ID of the process calling **setuid( )** is the super-user, the real, effective, and saved user IDs are set to the *uid* parameter.

If the effective user ID of the calling process is not the super-user, but *uid* is either the real user ID or the saved user ID of the calling process, the effective user ID is set to *uid.*

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

If the effective user ID of the calling process is not the super-user, but *gid* is either the real group ID or the saved group ID of the calling process, the effective group ID is set to *gid.*

RETURN VALUES

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

ERRORS

setuid() and **setgid()** fail if one or more of the following is true:

EINVAL The *uid* or *gid* is out of range.

EPERM For **setuid()** and **seteuid()** the effective user of the calling process is not super-user, and the *uid* parameter does not match either the real or saved user IDs. For **setgid()** and **setegid()** the effective user of the calling process is not the super-user, and the *gid* parameter does not match either the real or saved group IDs.

SEE ALSO

intro(2), **exec**(2), **getgroups**(2), **getuid**(2), **stat**(5)

NAME

shmctl – shared memory control operations

SYNOPSIS

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

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

DESCRIPTION

shmctl( ) provides a variety of shared memory control operations as specified by *cmd*. The permission required for a shared memory control operation is given as {*token*}, where *token* is the type of permission needed. The types of permission are interpreted as follows:

00400	READ by user
00200	WRITE by user
00040	READ by group
00020	WRITE by group
00004	READ by others
00002	WRITE by others

See the *Shared Memory Operation Permissions* section of **intro**(2) for more information.

The following operations require the specified tokens:

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 access permission bits */
```

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

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

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

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

RETURN VALUES

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

ERRORS

shmctl( ) fails if one or more of the following are true:

EACCES	*cmd* is equal to **IPC_STAT** and {READ} operation permission is denied to the calling process.
EFAULT	*buf* points to an illegal address.
EINVAL	*shmid* is not a valid shared memory identifier.
EINVAL	*cmd* is not a valid command.
EINVAL	*cmd* is **IPC_SET** and **shm_perm.uid** or **shm_perm.gid** is not valid.
ENOMEM	*cmd* is equal to **SHM_LOCK** and there is not enough memory.
EOVERFLOW	*cmd* is **IPC_STAT** and *uid* or *gid* is too large to be stored in the structure pointed to by *buf*.
EPERM	*cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user of the calling process is not super-user, or to the value of **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid*.
EPERM	*cmd* is equal to **SHM_LOCK** or **SHM_UNLOCK** and the effective user ID of the calling process is not equal to that of super-user.

SEE ALSO

ipcs(1), **intro**(2), **shmget**(2), **shmop**(2)

NOTES

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

NAME

shmget – get shared memory segment identifier

SYNOPSIS

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

int shmget(key_t key, int size, int 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 access permission bits of **shm_perm.mode** are set equal to the access permission 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.

RETURN VALUES

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

ERRORS

shmget( ) fails if one or more of the following are true:

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 both (*shmflg***&IPC_CREAT**) and (*shmflg***&IPC_EXCL**) are true.
EINVAL	*size* is less than the system-imposed minimum or greater than the system-imposed maximum.
EINVAL	A shared memory identifier exists for *key* but the size of the segment associated with it is less than *size* and *size* is not equal to zero.
ENOENT	A shared memory identifier does not exist for *key* and (*shmflg***&IPC_CREAT**) is false.

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

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

intro(2), **shmctl**(2), **shmop**(2), **stdipc**(3C)

NOTES

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

NAME

shmop, shmat, shmdt – shared memory operations

SYNOPSIS

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

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

int shmdt(void *shmaddr);
```

DESCRIPTION

shmat( ) attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process.

The permission required for a shared memory control operation is given as {*token*}, where *token* is the type of permission needed. The types of permission are interpreted as follows:

00400	READ by user
00200	WRITE by user
00040	READ by group
00020	WRITE by group
00004	READ by others
00002	WRITE by others

See the *Shared Memory Operation Permissions* section of **intro**(2) for more information.

When (*shmflg***&SHM_SHARE_MMU**) is true, virtual memory resources in addition to shared memory itself are shared among processes that use the same shared memory.

The shared memory segment is attached to the data segment of the calling process at the address specified based on one of the following criteria:

- If *shmaddr* is equal to **(void *) 0**, the segment is attached to the first available address as selected by the system.
- If *shmaddr* is equal to **(void *) 0** and (*shmflg***&SHM_SHARE_MMU**) is true, then the segment is attached to the first available aligned address. See **NOTES** for the alignment requirement.
- If *shmaddr* is not equal to **(void *) 0** and (*shmflg***&SHM_RND**) is true, the segment is attached to the address given by (*shmaddr* **-** (*shmaddr* modulus **SHMLBA**)).
- If *shmaddr* is not equal to **(void *) 0** and (*shmflg***&SHM_RND**) is false, the segment is attached to the address given by *shmaddr*.

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

When (*shmflg***&SHM_SHARE_MMU**) is set, however, the permission given by **shmget( )** determines whether the segment is attached for reading or reading and writing.

shmdt( ) detaches from the calling process's data segment the shared memory segment

located at the address specified by *shmaddr*.

RETURN VALUES

Upon successful completion, the return value is as follows:

shmat( ) returns the data segment start address of the attached shared memory segment.

shmdt( ) returns a value of 0.

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

ERRORS

shmat( ) fails and does not attach 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.
EINVAL	*shmaddr* is not equal to zero, and the value of (*shmaddr* **-** (*shmaddr* modulus **SHMLBA**)) is an illegal address.
EINVAL	*shmaddr* is not equal to zero, (*shmflg***&SHM_RND**) is false, and the value of *shmaddr* is an illegal address.
EINVAL	shmaddr is not equal to zero, (*shmfg***&SHM_SHARE_MMU**) is true, and *shmaddr* is not aligned properly.
EINVAL	**shmdt( )** fails and does not detach the shared memory segment if *shmaddr* is not the data segment start address of a shared memory segment.
EINVAL	**SHM_SHARE_MMU** is not supported in certain architectures.
EMFILE	The number of shared memory segments attached 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.

SEE ALSO

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

NOTES

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

The alignment requirement, which varies on different machines, is determined by the mapping size of the virtual memory system.

NAME

sigaction – detailed signal management

SYNOPSIS

#include <signal.h>

int sigaction(int *sig***, const struct sigaction** ∗*act***, struct sigaction** ∗*oact***);**

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

sigaction( ) allows the calling process to examine or specify the action to be taken on delivery of a specific signal. (See **signal**(5) for an explanation of general signal concepts.)

sig specifies the signal and can be assigned any of the signals specified in **signal**(5) except **SIGKILL** and **SIGSTOP.** In a multi-threaded process, *sig* cannot be **SIGWAITING, SIGCANCEL,** or **SIGLWP.**

If the argument *act* is not **NULL,** it points to a structure specifying the new action to be taken when delivering *sig*. If the argument *oact* is not **NULL,** it points to a structure where the action previously associated with *sig* is to be stored on return from **sigaction( )**.

The **sigaction** structure includes the following members:

```
void      (*sa_handler)( );
void      (*sa_sigaction)(int, siginfo_t *, void *);
sigset_t  sa_mask;
int       sa_flags;
```

sa_handler identifies the action to be associated with the specified signal, if the **SA_SIGINFO** flag (see below) is cleared in the **sa_flags** field of the sigaction structure. It may take any of the values specified in **signal**(5) or that of a user specified signal handler. If the **SA_SIGINFO** flag is set in the **sa_flags** field, the **sa_sigaction** field specifies a signal-catching function.

sa_mask specifies a set of signals to be blocked while the signal handler is active. On entry to the signal handler, that set of signals is added to the set of signals already being blocked when the signal is delivered. In addition, the signal that caused the handler to be executed will also be blocked, unless the **SA_NODEFER** flag has been specified. **SIGSTOP** and **SIGKILL** cannot be blocked (the system silently enforces this restriction).

sa_flags specifies a set of flags used to modify the delivery of the signal. It is formed by a logical OR of any of the following values:

SA_ONSTACK If set and the signal is caught, and if the LWP that is chosen to processes a delivered signal has an alternate signal stack declared with **sigaltstack**(2), then it will process the signal on that stack. Otherwise, the signal is delivered on the LWP main stack. Unbound threads (see **thr_create**(3T)) may not have alternate signal stacks.

SA_RESETHAND If set and the signal is caught, the disposition of the signal is reset to **SIG_DFL** and the signal will not be blocked on entry to the signal handler (**SIGILL**, **SIGTRAP**, and **SIGPWR** cannot be automatically reset when delivered; the

system silently enforces this restriction).

SA_NODEFER If set and the signal is caught, the signal will not be automatically blocked by the kernel while it is being caught.

SA_RESTART If set and the signal is caught, certain functions that are interrupted by the execution of this signal's handler are transparently restarted by the system; namely, **read**(2) or **write**(2) on slow devices like terminals, **ioctl**(2), **fcntl**(2), **wait**(2), and **waitid**(2). Otherwise, that function returns an **EINTR** error.

SA_SIGINFO If cleared and the signal is caught, *sig* is passed as the only argument to the signal-catching function. If set and the signal is caught, pending signals of type *sig* are reliably queued to the calling process and two additional arguments are passed to the signal-catching function. If the second argument is not equal to **NULL,** it points to a **siginfo_t** structure containing the reason why the signal was generated (see **siginfo**(5)); the third argument points to a **ucontext_t** structure containing the receiving process's context when the signal was delivered (see **ucontext**(5)).

SA_NOCLDWAIT If set and *sig* equals **SIGCHLD,** the system will not create zombie processes when children of the calling process exit. If the calling process subsequently issues a **wait**(2), it blocks until all of the calling process's child processes terminate, and then returns a value of –1 with **errno** set to **ECHILD.**

SA_NOCLDSTOP If set and *sig* equals **SIGCHLD, SIGCHLD** will not be sent to the calling process when its child processes stop or continue.

SA_WAITSIG If set and *sig* equals **SIGWAITING,** then the system will send **SIGWAITING** to the process when all the LWPs in the process are blocked.

RETURN VALUES

On success, **sigaction( )** returns zero. On failure, it returns –1 and sets **errno** to indicate the error. If **sigaction( )** fails, no new signal handler is installed.

ERRORS

sigaction( ) fails if any of the following is true:

EINVAL The value of the *sig* argument is not a valid signal number or is equal to **SIGKILL** or **SIGSTOP.** In addition, if in a multi-threaded process, it is equal to **SIGWAITING, SIGCANCEL,** or **SIGLWP.**

EFAULT *act* or *oact* points to an illegal address.

SEE ALSO

kill(1), **intro**(2), **exit**(2), **kill**(2), **pause**(2), **sigaltstack**(2), **sigprocmask**(2), **sigsend**(2), **sigsuspend**(2), **wait**(2), **signal**(3C), **sigsetops**(3C), **thr_create**(3T), **siginfo**(5), **signal**(5), **ucontext**(5)

NOTES

The handler routine can be declared:

void handler (int sig, siginfo_t *sip, ucontext_t *uap);

Here, **sig** is the signal number. **sip** is a pointer (to space on the stack) to a **siginfo_t** structure, which provides additional detail about the delivery of the signal. **uap** is a pointer (again to space on the stack) to a **ucontext_t** structure (defined in **sys/ucontext.h**) which contains the context from before the signal. It is not recommended that **uap** be used by the handler to restore the context from before the signal delivery.

NAME

sigaltstack – set or get signal alternate stack context

SYNOPSIS

```
#include <signal.h>
int sigaltstack(const stack_t *ss, stack_t *oss);
```

DESCRIPTION

sigaltstack( ) allows an LWP to define an alternate stack area on which signals are to be processed. If *ss* is non-zero, it specifies a pointer to, and the size of a stack area on which to deliver signals, and tells the system whether the LWP is currently executing on that stack. When a signal's action indicates its handler should execute on the alternate signal stack (specified with a **sigaction**(2) call), the system checks to see if the LWP chosen to execute the signal handler is currently executing on that stack. If the LWP is not currently executing on the signal stack, the system arranges a switch to the alternate signal stack for the duration of the signal handler's execution.

The structure **stack_t** includes the following members:

```
int     *ss_sp
long    ss_size
int     ss_flags
```

If *ss* is not **NULL**, it points to a structure specifying the alternate signal stack that will take effect upon successful return from **sigaltstack( )**. The **ss_sp** and **ss_size** fields specify the new base and size of the stack, which is automatically adjusted for direction of growth and alignment. The **ss_flags** field specifies the new stack state and may be set to the following:

SS_DISABLE The stack is to be disabled and **ss_sp** and **ss_size** are ignored. If **SS_DISABLE** is not set, the stack will be enabled.

If *oss* is not **NULL**, it points to a structure specifying the alternate signal stack that was in effect prior to the call to **sigaltstack( )**. The **ss_sp** and **ss_size** fields specify the base and size of that stack. The **ss_flags** field specifies the stack's state, and may contain the following values:

SS_ONSTACK The LWP is currently executing on the alternate signal stack. Attempts to modify the alternate signal stack while the LWP is executing on it will fail.

SS_DISABLE The alternate signal stack is currently disabled.

RETURN VALUES

On success, **sigaltstack( )** returns zero. On failure, it returns –1 and sets **errno** to indicate the error.

ERRORS

sigaltstack( ) fails if any of the following is true:

EFAULT *ss* or *oss* points to an illegal address.

EINVAL An attempt was made to disable an active stack or the **ss_flags** field in *ss* specifies invalid flags.

ENOMEM The size of the alternate stack area is less than **MINSIGSTKSZ**.

SEE ALSO

getcontext(2), **sigaction**(2), **ucontext**(5)

NOTES

The value **SIGSTKSZ** is defined to be the number of bytes that would be used to cover the usual case when allocating an alternate stack area. The value **MINSIGSTKSZ** is defined to be the minimum stack size for a signal handler. In computing an alternate stack size, a program should add that amount to its stack requirements to allow for the operating system overhead.

The following code fragment is typically used to allocate an alternate stack:

```
if ((sigstk.ss_sp = (char *)malloc(SIGSTKSZ)) == NULL)
        /* error return */;

sigstk.ss_size = SIGSTKSZ;
sigstk.ss_flags = 0;
if (sigaltstack(&sigstk, (stack_t *)0) < 0)
        perror("sigaltstack");
```

NAME

sigpending – examine signals that are blocked and pending

SYNOPSIS

```
#include <signal.h>
int sigpending(sigset_t *set);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

The **sigpending( )** function retrieves those signals that have been sent to the calling process but are being blocked from delivery by the calling process's signal mask. The signals are stored in the space pointed to by the argument *set*.

RETURN VALUES

On success, **sigpending( )** returns zero. On failure, it returns –1 and sets **errno** to indicate the error.

ERRORS

sigpending( ) fails if the following is true:

EFAULT *set* points to an illegal address.

SEE ALSO

sigaction(2), **sigprocmask**(2), **sigsetops**(3C)

NAME

sigprocmask – change and / or examine calling process's signal mask

SYNOPSIS

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

The **sigprocmask( )** function is used to examine and / or change the calling process's signal mask. If the value is **SIG_BLOCK**, the set pointed to by the argument *set* is added to the current signal mask. If the value is **SIG_UNBLOCK**, the set pointed by the argument *set* is removed from the current signal mask. If the value is **SIG_SETMASK**, the current signal mask is replaced by the set pointed to by the argument *set*. If the argument *oset* is not **NULL**, the previous mask is stored in the space pointed to by *oset*. If the value of the argument *set* is **NULL**, the value *how* is not significant and the process's signal mask is unchanged; 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 will be delivered before the call to **sigprocmask( )** returns.

It is not possible to block those signals that cannot be ignored (see **sigaction**(2)); this restriction is silently imposed by the system.

If **sigprocmask( )** fails, the process's signal mask is not changed.

RETURN VALUES

On success, **sigprocmask( )** returns zero. On failure, it returns –1 and sets **errno** to indicate the error.

ERRORS

sigprocmask( ) fails if any of the following is true:

EFAULT *set* or *oset* points to an illegal address.

EINVAL The value of the *how* argument is not equal to one of the defined values.

SEE ALSO

sigaction(2), **signal**(3C), **sigsetops**(3C), **signal**(5)

NAME

sigsend, sigsendset – send a signal to a process or a group of processes

SYNOPSIS

```
#include <signal.h>
int sigsend(idtype_t idtype, id_t id, int sig);
int sigsendset(procset_t *psp, int sig);
```

DESCRIPTION

sigsend( ) sends a signal to the process or group of processes specified by *id* and *idtype*. The signal to be sent is specified by *sig* and is either zero or one of the values listed in **signal**(5). If *sig* is zero (the null signal), error checking is performed but no signal is actually sent. This value can be used to check the validity of *id* and *idtype*.

The real or effective user ID of the sending process must match the real or effective user ID of the receiving process, unless the effective user ID of the sending process is superuser, or *sig* is **SIGCONT** and the sending process has the same session ID as the receiving process.

If *idtype* is **P_PID**, *sig* is sent to the process with process ID *id*.

If *idtype* is **P_PGID**, *sig* is sent to any process with process group ID *id*.

If *idtype* is **P_SID**, *sig* is sent to any process with session ID *id*.

If *idtype* is **P_UID**, *sig* is sent to any process with effective user ID *id*.

If *idtype* is **P_GID**, *sig* is sent to any process with effective group ID *id*.

If *idtype* is **P_CID**, *sig* is sent to any process with scheduler class ID *id* (see **priocntl**(2)).

If *idtype* is **P_ALL**, *sig* is sent to all processes and *id* is ignored.

If *id* is **P_MYID**, the value of *id* is taken from the calling process.

The process with a process ID of 0 is always excluded. The process with a process ID of 1 is excluded unless *idtype* is equal to **P_PID**.

sigsendset( ) provides an alternate interface for sending signals to sets of processes. This function sends signals to the set of processes specified by *psp*. *psp* is a pointer to a structure of type **procset_t**, defined in **<sys/procset.h>**, which includes the following members:

```
idop_t      p_op;
idtype_t    p_lidtype;
id_t        p_lid;
idtype_t    p_ridtype;
id_t        p_rid;
```

p_lidtype and **p_lid** specify the ID type and ID of one ("left") set of processes; **p_ridtype** and **p_rid** specify the ID type and ID of a second ("right") set of processes. ID types and IDs are specified just as for the *idtype* and *id* arguments to **sigsend()**. **p_op** specifies the operation to be performed on the two sets of processes to get the set of processes the function is to apply to. The valid values for **p_op** and the processes they specify are:

POP_DIFF set difference: processes in left set and not in right set

POP_AND set intersection: processes in both left and right sets

POP_OR set union: processes in either left or right set or both

POP_XOR set exclusive-or: processes in left or right set but not in both

RETURN VALUES

On success, **sigsend()** returns zero. On failure, it returns –1 and sets **errno** to indicate the error.

ERRORS

sigsend() and **sigsendset()** fail if one or more of the following are true:

EINVAL *sig* is not a valid signal number.

EINVAL *idtype* is not a valid idtype field.

EINVAL *sig* is **SIGKILL**, *idtype* is **P_PID** and *id* is 1 (proc1).

EPERM The effective user of the calling process is not super-user, and its real or effective user ID does not match the real or effective user ID of the receiving process, and the calling process is not sending **SIGCONT** to a process that shares the same session.

ESRCH No process can be found corresponding to that specified by *id* and *idtype*.

In addition, **sigsendset()** fails if:

EFAULT **psp** points to an illegal address.

SEE ALSO

kill(1), **getpid**(2), **kill**(2), **priocntl**(2), **signal**(3C), **signal**(5)

NAME

sigsuspend – install a signal mask and suspend process until signal

SYNOPSIS

```
#include <signal.h>
int sigsuspend(const sigset_t *set);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

sigsuspend( ) replaces the process's signal mask with the set of signals pointed to by the argument *set* 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. On return, the signal mask is restored to the set that existed before the call to **sigsuspend( )**.

It is not possible to block those signals that cannot be ignored (see **signal**(5)); this restriction is silently imposed by the system.

RETURN VALUES

Since **sigsuspend( )** suspends process execution indefinitely, there is no successful completion return value. On failure, it returns –1 and sets **errno** to indicate the error.

ERRORS

sigsuspend( ) fails if either of the following is true:

EFAULT *set* points to an illegal address.

EINTR A signal is caught by the calling process and control is returned from the signal catching function.

SEE ALSO

sigaction(2), **sigprocmask**(2), **signal**(3C), **sigsetops**(3C), **signal**(5)

NAME

sigwait – wait until a signal is posted

SYNOPSIS

```
#include <signal.h>
int sigwait(sigset_t *set);
```

POSIX

```
cc [ flag... ] file ... –D_POSIX_PTHREAD_SEMANTICS [ library... ]
#include <signal.h>
int sigwait(const sigset_t *set, int *sig);
```

DESCRIPTION

sigwait( ) selects a signal in *set* that is pending on the calling thread (see **thr_create**(3T)) or LWP. If no signal in *set* is pending, then **sigwait( )** blocks until a signal in *set* becomes pending. The selected signal is cleared from the set of signals pending on the calling thread or LWP and the number of the signal is returned. The selection of a signal in *set* is independent of the signal mask of the calling thread or LWP. This means a thread or LWP can synchronously wait for signals that are being blocked by the signal mask of the calling thread or LWP.

If **sigwait( )** is called on an ignored signal, then the occurrence of the signal will be ignored, even though **sigwait( )** was called for this signal. If more than one thread or LWP waits for the same signal, only one is unblocked when the signal arrives.

RETURN VALUES

Upon successful completion, **sigwait( )** returns a signal number. Otherwise, it returns a value of –1 and sets **errno** to indicate an error. Upon successful completion, the POSIX version of **sigwait( )** returns zero and stores the received signal number at the location pointed to by *sig*. Otherwise, it returns the error number.

ERRORS

If any of the following conditions are detected, **sigwait( )** fails and returns an error:

EINVAL *set* contains an unsupported signal number.

EFAULT *set* points to an invalid address.

SEE ALSO

sigaction(2), **sigpending**(2), **sigprocmask**(2), **sigsuspend**(2), **thr_create**(3T), **thr_sigsetmask**(3T), **signal**(5)

NOTES

sigwait( ) cannot be used to wait for signals that cannot be caught (see **sigaction**(2)). This restriction is silently imposed by the system.

In Solaris 2.4 and earlier releases, the call to **sigwait( )** from a multi-threaded process overrode the signal's ignore disposition; even if a signal's disposition was **SIG_IGN ,** a call to **sigwait( )** resulted in catching the signal, if generated. This is incorrect behavior from the standpoint of the POSIX 1003.1c spec.

In Solaris 2.5, the behavior of **sigwait( )** was corrected, so that it does not override the signal's ignore disposition. This change can cause applications that rely on the old behavior to break. Applications should employ **sigwait( )** as follows: Install a dummy signal handler, thereby changing the disposition from **SIG_IGN** to having a handler. Then, any calls to **sigwait( )** for this signal would catch it upon generation.

NAME

stat, lstat, fstat – get file status

SYNOPSIS

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

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

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

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

MT-LEVEL

stat( ) and **fstat( )** are Async-Signal-Safe

DESCRIPTION

stat( ) obtains information about the file pointed to 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( ) obtains file attributes similar to **stat( )**, except when the named file is a symbolic link; in that case **lstat( )** returns information about the link, while **stat( )** returns information about the file the link references.

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

buf is a pointer to a **stat( )** structure into which information is placed concerning the file.

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

```
mode_t    st_mode;    /* File mode (see mknod(2)) */
ino_t     st_ino;     /* Inode number */
dev_t     st_dev;     /* ID of device containing */
                      /* a directory entry for this file */
dev_t     st_rdev;    /* ID of device */
                      /* This entry is defined only for */
                      /* char special or block special files */
nlink_t   st_nlink;   /* Number of links */
uid_t     st_uid;     /* User ID of the file's owner */
gid_t     st_gid;     /* Group ID of the file's group */
off_t     st_size;    /* File size in bytes */
time_t    st_atime;   /* Time of last access */
time_t    st_mtime;   /* Time of last data modification */
time_t    st_ctime;   /* Time of last file status change */
                      /* Times measured in seconds since */
                      /* 00:00:00 UTC, Jan. 1, 1970 */
long      st_blksize; /* Preferred I/O block size */
long      st_blocks;  /* Number of 512 byte blocks allocated*/
```

st_mode The mode of the file as described in **mknod**(2). In addition to the modes described in **mknod**(2), the mode of a file may also be S_IFLNK if the file is a symbolic link. (Note that S_IFLNK may only be returned by **lstat( )**.)

st_ino This field uniquely identifies the file in a given file system. The pair **st_ino**

and **st_dev** uniquely identifies regular files.

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

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

st_nlink This field should be used only by administrative commands.

st_uid The user ID of the file's owner.

st_gid The group ID of the file's group.

st_size For regular files, this is the address of the end of the file. For block special or character special, this is not defined. See also **pipe**(2).

st_atime Time when file data was last accessed. Changed by the following functions: **creat**, **mknod**, **pipe**, **utime**, and **read**.

st_mtime Time when data was last modified. Changed by the following functions: **creat**, **mknod**, **pipe**, **utime**, and **write**.

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

st_blksize A hint as to the "best" unit size for I/O operations. This field is not defined for block special or character special files.

st_blocks The total number of physical blocks of size 512 bytes actually allocated on disk. This field is not defined for block special or character special files.

RETURN VALUES

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

ERRORS

stat( ) and **lstat( )** fail if one or more of the following are true:

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

EFAULT *buf* or *path* points to an illegal address.

EINTR A signal was caught during the **stat( )** or **lstat( )** function.

ELOOP Too many symbolic links were encountered in translating *path*.

EMULTIHOP Components of *path* require hopping to multiple remote machines and the file system does not allow it.

ENAMETOOLONG The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.

ENOENT The named file does not exist or is the null pathname.

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

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

EOVERFLOW	A component is too large to store in the structure pointed to by *buf*.

fstat( ) fails if one or more of the following are true:

EBADF	*fildes* is not a valid open file descriptor.
EFAULT	*buf* points to an illegal address.
EINTR	A signal was caught during the **fstat( )** function.
ENOLINK	*fildes* points to a remote machine and the link to that machine is no longer active.
EOVERFLOW	A component is too large to store in the structure pointed to by *buf*.

SEE ALSO **chmod**(2), **chown**(2), **creat**(2), **link**(2), **mknod**(2), **pipe**(2), **read**(2), **time**(2), **unlink**(2), **utime**(2), **write**(2), **fattach**(3C), **stat**(5)

NAME

statvfs, fstatvfs – get file system information

SYNOPSIS

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

int statvfs(const char *path, struct statvfs *buf);

int fstatvfs(int fildes, struct statvfs *buf);
```

DESCRIPTION

statvfs( ) returns a "generic superblock" describing a file system; it can be used to acquire information about mounted file systems. *buf* is a pointer to a structure (described below) that is filled by the function.

path should name a file that resides on that file system. The file system type is known to the operating system. Read, write, or execute permission for the named file is not required, but all directories listed in the path name leading to the file must be searchable.

The **statvfs( )** structure pointed to by *buf* includes the following members:

```
u_long   f_bsize;              /* preferred file system block size */
u_long   f_frsize;             /* fundamental filesystem block size
                                  (if supported) */
u_long   f_blocks;             /* total # of blocks on file system
                                  in units of f_frsize */
u_long   f_bfree;              /* total # of free blocks */
u_long   f_bavail;             /* # of free blocks avail to
                                  non-super-user */
u_long   f_files;              /* total # of file nodes (inodes) */
u_long   f_ffree;              /* total # of free file nodes */
u_long   f_favail;             /* # of inodes avail to
                                  non-super-user*/
u_long   f_fsid;               /* file system id (dev for now) */
char     f_basetype[FSTYPSZ];  /*  target fs type name,
                                  null-terminated */
u_long   f_flag;               /* bit mask of flags */
u_long   f_namemax;            /* maximum file name length */
char     f_fstr[32];           /* file system specific string */
u_long   f_filler[16];         /* reserved for future expansion */
```

f_basetype contains a null-terminated FSType name of the mounted target.

The following flags can be returned in the **f_flag** field:

```
ST_RDONLY    0x01    /* read-only file system */
ST_NOSUID    0x02    /* does not support setuid/setgid
                        semantics */
ST_NOTRUNC   0x04    /* does not truncate file names
                        longer than {NAME_MAX}*/
```

fstatvfs() is similar to **statvfs()**, except that the file named by *path* in **statvfs()** is instead identified by an open file descriptor *fildes* obtained from a successful **open**(2), **creat**(2), **dup**(2), **fcntl**(2), or **pipe**(2) function.

RETURN VALUES

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

ERRORS

statvfs() fails if one or more of the following are true:

EACCES	Search permission is denied on a component of the path prefix.
EFAULT	*path* or *buf* points to an illegal address.
EINTR	A signal was caught during **statvfs()** execution.
EIO	An I/O error occurred while reading the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines and file system type does not allow it.
ENAMETOOLONG	The length of a *path* component exceeds {**NAME_MAX**} characters, or the length of *path* exceeds {**PATH_MAX**} characters.
ENOENT	Either a component of the path prefix or the file referred to by *path* does not exist.
ENOLINK	*path* points to a remote machine and the link to that machine is no longer active.
ENOTDIR	A component of the path prefix of *path* is not a directory.

fstatvfs() fails if one or more of the following are true:

EBADF	*fildes* is not an open file descriptor.
EFAULT	*buf* points to an illegal address.
EINTR	A signal was caught during **fstatvfs()** execution.
EIO	An I/O error occurred while reading the file system.

SEE ALSO

chmod(2), **chown**(2), **creat**(2), **dup**(2), **fcntl**(2), **link**(2), **mknod**(2), **open**(2), **pipe**(2), **read**(2), **time**(2), **unlink**(2), **utime**(2), **write**(2)

BUGS

The values returned for **f_files**, **f_ffree**, and **f_favail** may not be valid for NFS mounted file systems.

NAME

stime – set system time and date

SYNOPSIS

```
#include <unistd.h>
int stime(const time_t *tp);
```

DESCRIPTION

stime( ) sets the system's idea of the time and date. *tp* points to the value of time as measured in seconds from 00:00:00 UTC January 1, 1970.

RETURN VALUES

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

ERRORS

stime( ) will fail if:

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

SEE ALSO

time(2)

NAME

swapctl – manage swap space

SYNOPSIS

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

int swapctl(int cmd, void *arg);
```

DESCRIPTION

swapctl( ) adds, deletes, or returns information about swap resources. *cmd* specifies one of the following options contained in **<sys/swap.h>**:

```
SC_ADD          /* add a resource for swapping */
SC_LIST         /* list the resources for swapping */
SC_REMOVE       /* remove a resource for swapping */
SC_GETNSWP      /* return number of swap resources */
```

When **SC_ADD** or **SC_REMOVE** is specified, *arg* is a pointer to a **swapres** structure containing the following members:

```
char    *sr_name;       /* pathname of resource */
off_t   sr_start;       /* offset to start of swap area */
off_t   sr_length;      /* length of swap area */
```

sr_start and **sr_length** are specified in 512-byte blocks.

When **SC_LIST** is specified, *arg* is a pointer to a **swaptable** structure containing the following members:

```
int     swt_n;          /* number of swapents following */
struct  swapent swt_ent[];  /* array of swt_n swapents */
```

A **swapent** structure contains the following members:

```
char    *ste_path;      /* name of the swap file */
off_t   ste_start;      /* starting block for swapping */
off_t   ste_length;     /* length of swap area */
long    ste_pages;      /* number of pages for swapping */
long    ste_free;       /* number of ste_pages free */
long    ste_flags;      /* ST_INDEL bit set if swap file */
                        /* is now being deleted */
```

SC_LIST causes **swapctl( )** to return at most **swt_n** entries. The return value of **swapctl( )** is the number actually returned. The **ST_INDEL** bit is turned on in **ste_flags** if the swap file is in the process of being deleted.

When **SC_GETNSWP** is specified, **swapctl( )** returns as its value the number of swap resources in use. *arg* is ignored for this operation.

The **SC_ADD** and **SC_REMOVE** functions will fail if calling process does not have appropriate privileges.

RETURN VALUES

Upon successful completion, the function **swapctl( )** returns a value of **0** for **SC_ADD** or **SC_REMOVE**, the number of **struct swapent** entries actually returned for **SC_LIST**, or the number of swap resources in use for **SC_GETNSWP**. Upon failure, the function **swapctl( )** returns a value of **–1** and sets **errno** to indicate an error.

ERRORS

Under the following conditions, the function **swapctl( )** fails and sets **errno** to:

EEXIST	Part of the range specified by **sr_start** and **sr_length** is already being used for swapping on the specified resource (**SC_ADD**).
EFAULT	*arg*, **sr_name**, or **ste_path** points to an illegal address.
EINVAL	The specified function value is not valid, the path specified is not a swap resource (**SC_REMOVE**), part of the range specified by **sr_start** and **sr_length** lies outside the resource specified (**SC_ADD**), or the specified swap area is less than one page (**SC_ADD**).
EISDIR	The path specified for **SC_ADD** is a directory.
ELOOP	Too many symbolic links were encountered in translating the pathname provided to **SC_ADD** or **SC_REMOVE** .
ENAMETOOLONG	The length of a component of the path specified for **SC_ADD** or **SC_REMOVE** exceeds **{NAME_MAX}** characters or the length of the path exceeds **{PATH_MAX}** characters and **{_POSIX_NO_TRUNC}** is in effect.
ENOENT	The pathname specified for **SC_ADD** or **SC_REMOVE** does not exist.
ENOMEM	An insufficient number of **struct swapent** structures were provided to **SC_LIST**, or there were insufficient system storage resources available during an **SC_ADD** or **SC_REMOVE**, or the system would not have enough swap space after an **SC_REMOVE**.
ENOSYS	The pathname specified for **SC_ADD** or **SC_REMOVE** is not a file or block special device.
ENOTDIR	Pathname provided to **SC_ADD** or **SC_REMOVE** contained a component in the path prefix that was not a directory.
EPERM	The effective user of the calling process is not super-user.
EROFS	The pathname specified for **SC_ADD** is a read-only file system.

NAME

symlink – make a symbolic link to a file

SYNOPSIS

#include <unistd.h>

int symlink(const char **name1***, const char** **name2***);**

DESCRIPTION

symlink( ) creates a symbolic link *name2* to the file *name1*. Either name may be an arbitrary pathname, the files need not be on the same file system, and *name1* may be nonexistent.

The file to which the symbolic link points is used when an **open**(2) operation is performed on the link. A **stat**(2) on a symbolic link returns the linked-to file, while an **lstat** 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

Upon successful completion **symlink( )** returns a value of 0; otherwise, it returns –1 and places an error code in **errno**.

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*.
EEXIST	The file referred to by *name2* already exists.
EFAULT	*name1* or *name2* points to an illegal address.
EIO	An I/O error occurs while reading from or writing to the file system.
ELOOP	Too many symbolic links are encountered in translating *name2*.
ENAMETOOLONG	The length of the *name2* argument exceeds {**PATH_MAX**}, or the length of a *name2* component exceeds {**NAME_MAX**} while (**_POSIX_NO_TRUNC**) is in effect.
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 no space is left on the file system containing the directory.
ENOSPC	The new symbolic link cannot be created because no space is left on the file system which will contain the link.
ENOSPC	There are no free inodes on the file system on which the file is being created.

ENOSYS	The file system does not support symbolic links
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 **cp**(1), **link**(2), **open**(2), **readlink**(2), **stat**(2), **unlink**(2)

NAME

sync – update super block

SYNOPSIS

```
#include <unistd.h>
void sync(void);
```

DESCRIPTION

sync( ) causes all information in memory that should be on disk to be written out. This includes modified super blocks, modified i-nodes, and delayed block I / O.

It should be used by programs that examine a file system, such as **fsck**(1M), **df**(1M), etc. It is mandatory before a re-boot.

The writing, although scheduled, is not necessarily completed before **sync( )** returns. The **fsync** function completes the writing before it returns.

SEE ALSO

df(1M), **fsck**(1M), **fsync**(3C)

NAME

sysfs – get file system type information

SYNOPSIS

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

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

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

int sysfs(int opcode);
```

DESCRIPTION

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

GETFSIND Translate *fsname*, a null-terminated file-system type identifier, into a file-system type index.

GETFSTYP Translate *fs_index*, a file-system type index, into a null-terminated file-system type identifier and write it into the buffer pointed to by *buf*; this buffer must be at least of size **FSTYPSZ** as defined in **<sys/fstyp.h>**.

GETNFSTYP Return the total number of file system types configured in the system.

RETURN VALUES

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

ERRORS

sysfs( ) fails if one or more of the following are true:

EFAULT *buf* or *fsname* points to an illegal address.

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

NAME

sysinfo – get and set system information strings

SYNOPSIS

#include <sys/systeminfo.h>

long sysinfo(int *command***, char** **buf***, long** *count***);**

DESCRIPTION

sysinfo( ) copies information relating to the operating system on which the process is executing into the buffer pointed to by *buf*. **sysinfo( )** can also set certain information where appropriate *commands* are available. *count* is the size of the buffer.

The POSIX P1003.1 interface **sysconf**(3C) provides a similar class of configuration information, but returns an integer rather than a string.

The *command*s available are:

SI_SYSNAME Copy into the array pointed to by *buf* the string that would be returned by **uname**(2) in the *sysname* field. This is the name of the implementation of the operating system, for example, **SunOS** or **UTS**.

SI_HOSTNAME Copy into the array pointed to by *buf* a string that names the present host machine. This is the string that would be returned by **uname**(2) in the *nodename* field. This hostname or nodename is often the name the machine is known by locally.

The *hostname* is the name of this machine as a node in some network. Different networks may have different names for the node, but presenting the nodename to the appropriate network directory or name-to-address mapping service should produce a transport end point address. The name may not be fully qualified.

Internet host names may be up to **256** bytes in length (plus the terminating null).

SI_SET_HOSTNAME Copy the null-terminated contents of the array pointed to by *buf* into the string maintained by the kernel whose value will be returned by succeeding calls to **sysinfo( )** with the command **SI_HOSTNAME**. This command requires that the effective-user-id be super-user.

SI_RELEASE Copy into the array pointed to by *buf* the string that would be returned by **uname**(2) in the *release* field. Typical values might be **5.2** or **4.1**.

SI_VERSION Copy into the array pointed to by *buf* the string that would be returned by **uname**(2) in the *version* field. The syntax and semantics of this string are defined by the system provider.

SI_MACHINE Copy into the array pointed to by *buf* the string that would be returned by **uname**(2) in the *machine* field, for example, **sun4c**, **sun4d**, or **sun4m**.

SI_ARCHITECTURE Copy into the array pointed to by *buf* a string describing the instruction set architecture of the current system, for example, **sparc**, **mc68030**, **m32100**, or **i386**. These names may not match predefined names in the C language compilation system.

SI_PLATFORM Copy into the array pointed to by *buf* a string describing the specific model of the hardware platform, for example, **SUNW,Sun_4_75**, **SUNW,SPARCsystem-600**, or **i86pc**.

SI_HW_PROVIDER Copies the name of the hardware manufacturer into the array pointed to by *buf*.

SI_HW_SERIAL Copy into the array pointed to by *buf* a string which is the ASCII representation of the hardware-specific serial number of the physical machine on which the function is executed. Note that this may be implemented in Read-Only Memory, using software constants set when building the operating system, or by other means, and may contain non-numeric characters. It is anticipated that manufacturers will not issue the same "serial number" to more than one physical machine. The pair of strings returned by **SI_HW_PROVIDER** and **SI_HW_SERIAL** is likely to be unique across all vendor's SVR4 implementations.

SI_SRPC_DOMAIN Copies the Secure Remote Procedure Call domain name into the array pointed to by *buf*.

SI_SET_SRPC_DOMAIN
Set the string to be returned by **sysinfo()** with the **SI_SRPC_DOMAIN** command to the value contained in the array pointed to by *buf*. This command requires that the effective-user-id be super-user.

RETURN VALUES

Upon successful completion, the value returned indicates the buffer size in bytes required to hold the complete value and the terminating null character. If this value is no greater than the value passed in *count*, the entire string was copied. If this value is greater than *count*, the string copied into *buf* has been truncated to *count –1* bytes plus a terminating null character.

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

ERRORS

sysinfo() will fail if one or more of the following are true:

EFAULT *buf* does not point to a valid address.

EINVAL The data for a SET command exceeds the limits established by the implementation.

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

USAGE

In many cases there is no corresponding programmatic interface to set these values; such strings are typically settable only by the system administrator modifying entries in the **/etc/system** directory or the code provided by the particular OEM reading a serial number or code out of read-only memory, or hard-coded in the version of the operating system.

A good starting guess for *count* is 257, which is likely to cover all strings returned by this interface in typical installations.

SEE ALSO

uname(2), **gethostid**(3C), **gethostname**(3C), **sysconf**(3C)

NAME

time – get time

SYNOPSIS

```
#include <sys/types.h>
#include <time.h>
time_t time(time_t *tloc);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

time() returns the value of time in seconds since 00:00:00 UTC, January 1, 1970.

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

RETURN VALUES

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

SEE ALSO

stime(2), **ctime**(3C)

NOTES

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

NAME

times – get process and child process times

SYNOPSIS

```
#include <sys/times.h>
#include <limits.h>

clock_t times(struct tms *buffer);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

times( ) fills the **tms** structure pointed to by *buffer* with time-accounting information. The **tms** structure, defined in **<sys/times.h>**, contains the following members:

```
clock_t    tms_utime;
clock_t    tms_stime;
clock_t    tms_cutime;
clock_t    tms_cstime;
```

This information comes from the calling process and each of its terminated child processes for which it has executed a wait routine. All times are reported in clock ticks. The specific value for a clock tick is defined by the variable **CLK_TCK**, found in the header **<limits.h>**.

tms_utime is the CPU time used while executing instructions in the user space of the calling process.

tms_stime is the CPU time used by the system on behalf of the calling process.

tms_cutime is the sum of the **tms_utime** and the **tms_cutime** of the child processes.

tms_cstime is the sum of the **tms_stime** and the **tms_cstime** of the child processes.

RETURN VALUES

Upon successful completion, **times( )** returns the elapsed real time, in clock ticks, from an arbitrary point in the past (for example, system start-up time). This point does not change from one invocation of **times( )** to another. If **times( )** fails, a value of –1 is returned and **errno** is set to indicate the error.

ERRORS

times( ) fails if:

EFAULT *buffer* points to an illegal address.

SEE ALSO

time(1), **timex**(1), **exec**(2), **fork**(2), **time**(2), **wait**(2), **waitid**(2), **waitpid**(2)

NAME

uadmin – administrative control

SYNOPSIS

```
#include <sys/uadmin.h>
int uadmin(int cmd, int fcn, int mdep);
```

DESCRIPTION

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

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

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

- **AD_HALT** Halt the processor(s).
- **AD_POWEROFF** Halt the processor(s) and turn off the power.
- **AD_BOOT** Reboot the system, using the kernel file.
- **AD_IBOOT** Interactive reboot; user is prompted for bootable program name.

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

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

A_FREEZE Suspend the whole system. The system state is preserved in the state file. The following three subcommands are available.

- **AD_COMPRESS** Save the system state to the state file with compression of data.
- **AD_CHECK** Check if your system supports suspend and resume. Without performing a system suspend/resume, this command checks if this feature is currently available on your system.
- **AD_FORCE** Force **AD_COMPRESS** even when threads of drivers are not suspendable.

RETURN VALUES

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

A_SHUTDOWN Never returns.

A_REBOOT Never returns.

A_FREEZE **0** upon resume.

A_REMOUNT **0**

Upon unsuccessful completion, **–1** is returned and **errno** is set to indicate the error.

ERRORS

uadmin() fails if any of the following are true:

EPERM	The effective user of the calling process is not super-user.
ENOMEM	Suspend / resume ran out of physical memory.
ENOSPC	Suspend / resume could not allocate enough space on the root file system to store system information.
ENOTSUP	Suspend / resume not supported on this platform.
ENXIO	Unable to successfully suspend system.
EBUSY	Suspend already in progress.

SEE ALSO

kernel(1M), **uadmin**(1M)

NAME

ulimit – get and set process limits

SYNOPSIS

#include <ulimit.h>

long ulimit(int *cmd***,** */* newlimit */* **...);**

DESCRIPTION

This function provides for control over process limits. The *cmd* values available are:

UL_GETFSIZE Get the regular file size limit of the process. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

UL_SETFSIZE Set the regular file size limit of the process to the value of *newlimit* , taken as a **long**. Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit.

UL_GMEMLIM
Get the maximum possible break value (see **brk**(2)).

UL_GDESLIM Get the current value of the maximum number of open files per process configured in the system.

The **getrlimit( )** and **setrlimit( )** functions provide a more general interface for controlling process limits.

RETURN VALUES

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

ERRORS

ulimit( ) fails if the following is true:

EINVAL The *cmd* argument is not valid.

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

SEE ALSO

brk(2), **getrlimit**(2), **write**(2)

NOTES

ulimit( ) is effective in limiting the growth of regular files. Pipes are limited to {**PIPE_MAX**} bytes.

NAME

umask – set and get file creation mask

SYNOPSIS

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

mode_t umask(mode_t cmask);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

umask() sets the process's file mode creation mask to *cmask* and returns the previous value of the mask. Only the access permission bits of *cmask* and the file mode creation mask are used.

The mask is inherited by child processes.

See **intro**(2) for more information on masks.

RETURN VALUES

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

SEE ALSO

mkdir(1), **sh**(1), **intro**(2), **chmod**(2), **creat**(2), **mknod**(2), **open**(2), **stat**(5)

NAME

umount – unmount a file system

SYNOPSIS

```
#include <sys/mount.h>
int umount(const char *file);
```

DESCRIPTION

umount( ) requests that a previously mounted file system contained on the block special device or directory identified by *file* be unmounted. *file* is a pointer to a path name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

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

RETURN VALUES

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

ERRORS

umount( ) will fail if one or more of the following are true:

EBUSY	A file on *file* is busy.
EFAULT	*file* points to an illegal address.
EINVAL	*file* is not mounted.
ENOENT	*file* does not exist.
ELOOP	Too many symbolic links were encountered in translating the path pointed to by *file*.
EMULTIHOP	Components of the path pointed to by *file* require hopping to multiple remote machines.
ENAMETOOLONG	The length of the *file* argument exceeds {**PATH_MAX**}, or the length of a *file* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOLINK	*file* is on a remote machine, and the link to that machine is no longer active.
ENOTBLK	*file* is not a block special device.
EPERM	The process's effective user ID is not super-user.
EREMOTE	*file* is remote.

SEE ALSO

mount(2)

NAME

uname – get name of current operating system

SYNOPSIS

```
#include <sys/utsname.h>
int uname(struct utsname *name);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

uname( ) stores information identifying the current operating system in the structure pointed to by *name*.

uname( ) uses the structure **utsname** defined in **<sys/utsname.h>** whose members include:

```
char    sysname[SYS_NMLN];
char    nodename[SYS_NMLN];
char    release[SYS_NMLN];
char    version[SYS_NMLN];
char    machine[SYS_NMLN];
```

uname( ) returns a null-terminated character string naming the current operating system in the character array *sysname*. Similarly, *nodename* contains the name that the system is known by on a communications network. *release* and *version* further identify the operating system. *machine* contains a standard name that identifies the hardware that the operating system is running on.

RETURN VALUES

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

ERRORS

EFAULT **uname( )** fails if *name* points to an illegal address.

SEE ALSO

uname(1), **sysinfo**(2), **sysconf**(3C)

NAME

unlink – remove directory entry

SYNOPSIS

#include <unistd.h>

int unlink(const char **path***);**

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

unlink() removes the directory entry named by the path name pointed to by *path*. and decrements the link count of the file referenced by the directory entry. When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, space occupied by the file is not released until all references to the file have been closed. If *path* is a symbolic link, the symbolic link is removed. *path* should not name a directory unless the process has appropriate privileges. Applications should use **rmdir** to remove directories.

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

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

ERRORS

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

EACCES	Search permission is denied for a component of the *path* prefix.
EACCES	Write permission is denied on the directory containing the link to be removed.
EACCES	The parent directory has the sticky bit set and the file is not writable by the user; the user does not own the parent directory and the user does not own the file.
EBUSY	The entry to be unlinked is the mount point for a mounted file system.
EFAULT	*path* points to an illegal address.
EINTR	A signal was caught during the **unlink()** function.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines and the file system does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	The named file does not exist or is a null pathname.
ENOLINK	*path* points to a remote machine and the link to that machine is no

	longer active.
ENOTDIR	A component of the *path* prefix is not a directory.
EPERM	The named file is a directory and the effective user of the calling process is not super-user.
EROFS	The directory entry to be unlinked is part of a read-only file system.

SEE ALSO **rm**(1), **close**(2), **link**(2), **open**(2), **rmdir**(2)

NAME

ustat – get file system statistics

SYNOPSIS

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

int ustat(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 (see **makedev**(3C)). *buf* is a pointer to a **ustat( )** structure that includes the following elements:

```
daddr_t  f_tfree;     /* Total free blocks */
ino_t    f_tinode;    /* Number of free inodes */
char     f_fname[6];  /* Filsys name */
char     f_fpack[6];  /* Filsys pack name */
```

The last two fields, f_fname and f_fpack may not have significant information on all systems, and in that case, will contain the null character as the first character of these fields.

RETURN VALUES

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

ERRORS

ustat( ) fails if one or more of the following are true:

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

EFAULT *buf* points to an illegal address.

EINTR A signal was caught during a **ustat( )** function.

EINVAL *dev* is not the device number of a device containing a mounted file system.

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

SEE ALSO

stat(2), **statvfs**(2), **makedev**(3C)

NOTES

ustat( ) will be phased out in favor of the **statvfs**(2) function.

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

utime – set file access and modification times

SYNOPSIS

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

int utime(const char *path, const struct utimbuf *times);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

utime( ) sets the access and modification times of the file pointed to by *path*.

If *times* is **NULL**, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use **utime( )** in this manner.

If *times* is not **NULL,** *times* is interpreted as a pointer to a **utimbuf** structure (defined in **utime.h**) and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the super-user may use **utime( )** this way. The **utimbuf** structure contains the following members:

```
time_t actime;    /* access time */
time_t modtime;   /* modification time */
```

The times in the members of the **utimbuf** structure are measured in seconds since 00:00:00 UTC, Jan. 1, 1970.

utime( ) also causes the time of the last file status change (**st_ctime**) to be updated.

RETURN VALUES

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

ERRORS

utime( ) will fail if one or more of the following are true:

EACCES	Search permission is denied by a component of the *path* prefix.
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 *times* is NULL.
EFAULT	*path* points to an illegal address.
EINTR	A signal was caught during the **utime( )** function.
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*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines and the file system does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	The named file does not exist or is a null pathname.

ENOLINK	*path* points to a remote machine and the link to that machine is no longer active.
ENOTDIR	A component of the *path* prefix is not a directory.
EPERM	The effective user of the calling process is not super-user and not the owner of the file, and *times* is not **NULL**.
EROFS	The file system containing the file is mounted read-only.

SEE ALSO

stat(2)

NAME

utimes – set file times

SYNOPSIS

```
#include <sys/types.h>
#include <sys/time.h>
int utimes(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.

utimes( ) also causes the time of the last file status change (**st_ctime**) to be updated.

RETURN VALUES

utimes( ) returns:

0 on success.

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

ERRORS

utimes( ) will fail if one or more of the following are true:

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 to an illegal address.
EINTR	A signal was caught during the **utimes( )** function.
EINVAL	The number of microseconds specified in one or both of the **timeval** structures pointed to by *tvp* was greater than or equal to 1,000,000 or less than 0.
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*.
EMULTIHOP	Components of *file* require hopping to multiple remote machines and the file system does not allow it.
ENAMETOOLONG	The length of the *file* argument exceeds {**PATH_MAX**}, or the length of a path component of *file* exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.

ENOENT	The named file does not exist or is a null pathname.
ENOLINK	*file* points to a remote machine and the link to that machine is no longer active.
ENOTDIR	A component of the path prefix of *file* is not a directory.
EPERM	The effective user of the calling 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**(2)

NAME

vfork – spawn new process in a virtual memory efficient way

SYNOPSIS

```
#include <unistd.h>
pid_t vfork(void);
```

DESCRIPTION

vfork( ) can be used to create new processes without fully copying the address space of the old process. It is useful when the purpose of **fork( )** would have been to create a new system context for an **execve( )**. **vfork( )** differs from **fork( )** in that the child borrows the parent's memory and thread of control until a call to **execve( )** or an exit (either by a call to **_exit( )** (see **exit**(2)) 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**(3C) if you cannot **execve( )**, since **exit**(3C) will flush and close standard I/O channels, and thereby corrupt the parent processes standard I/O data structures. Even with **fork( )** it is wrong to call **exit**(3C) since buffered data would then be flushed twice.

RETURN VALUES

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

ERRORS

vfork( ) 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.

EAGAIN 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

exec(2), **exit**(2), **fork**(2), **ioctl**(2), **wait**(2), **exit**(3C)

NOTES

vfork( ) is unsafe in multi-thread applications.

This function will be eliminated in a future release. 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.

On some systems, the implementation of **vfork( )** causes the parent to inherit register values from the child. This can create problems for certain optimizing compilers if **<unistd.h>** is not included in the source calling **vfork( )**.

NAME

vhangup – virtually "hangup" the current controlling terminal

SYNOPSIS

void vhangup(void);

DESCRIPTION

vhangup( ) is used by the initialization process **init**(1M) (among others) to arrange that users are given "clean" terminals at login, by revoking access of the previous users' processes to the terminal. To effect this, **vhangup( )** searches the system tables for references to the controlling 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 or EIO). Finally, a **SIGHUP** (hangup signal) is sent to the process group of the controlling terminal.

SEE ALSO

init(1M)

BUGS

Access to the controlling terminal using **/dev/tty** is still possible.

This call should be replaced by an automatic mechanism that takes place on process exit.

NAME

wait – wait for child process to stop or terminate

SYNOPSIS

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

pid_t wait(int *stat_loc);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

wait( ) suspends the calling process until one of its immediate children terminates or until a child that is being traced stops because it has received a signal. The **wait( )** function will return prematurely if a signal is received. If any unawaited process stopped or terminated prior to the call on **wait( )**, return is immediate.

If **wait( )** returns because the status of a child process is available, it returns the process ID of the child process. If the calling process had specified a non-zero value for *stat_loc*, the status of the child process will be stored in the location pointed to by *stat_loc*. It may be evaluated with the macros described on **wstat**(5). In the following, *status* is the object pointed to by *stat_loc*:

If the child process stopped, the high order 8 bits of *status* will contain the number of the signal that caused the process to stop and the low order 8 bits will be set equal to **WSTOPFLG**.

If the child process terminated due to an **_exit( )** call, the low order 8 bits of *status* will be 0 and the high order 8 bits will contain the low order 8 bits of the argument that the child process passed to **_exit( );** see **exit**(2).

If the child process terminated due to a signal, the high order 8 bits of *status* will be 0 and the low order 8 bits will contain the number of the signal that caused the termination. In addition, if **WCOREFLG** is set, a “core image” will have been produced; see **signal**(3C).

If **wait( )** returns because the status of a child process is available, then that status may be evaluated with the macros defined by **wstat**(5).

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes; see **intro**(2).

RETURN VALUES

When **wait( )** returns due to a 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.

ERRORS

wait( ) will fail if one or both of the following is true:

ECHILD	The calling process has no existing unwaited-for child processes.
EINTR	The function was interrupted by a signal.

SEE ALSO **intro**(2), **exec**(2), **exit**(2), **fork**(2), **pause**(2), **ptrace**(2), **waitid**(2), **waitpid**(2), **signal**(3C), **signal**(5), **wstat**(5)

NOTES See NOTES in **signal**(3C).

Since **wait( )** will block on a stopped child, if the calling process wishes to see the return results of such a **wait**, it should use **waitid**(2) or **waitpid**(2) instead of **wait( )**.

NAME

waitid – wait for child process to change state

SYNOPSIS

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

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

DESCRIPTION

waitid( ) suspends the calling process until one of its children changes state. It records the current state of a child in the structure pointed to by *infop*. If a child process changed state prior to the call to **waitid( )**, **waitid( )** returns immediately.

The *idtype* and *id* arguments specify which children **waitid( )** is to wait for.

If *idtype* is **P_PID**, **waitid( )** waits for the child with a process ID equal to **(pid_t)** *id*.

If *idtype* is **P_PGID**, **waitid( )** waits for any child with a process group ID equal to **(pid_t)***id*.

If *idtype* is **P_ALL**, **waitid( )** waits for any child and *id* is ignored.

The *options* argument is used to specify which state changes **waitid( )** is to wait for. It is formed by an OR of any of the following flags:

WCONTINUED	Return the status for any child that was stopped and has been continued.
WEXITED	Wait for process(es) to exit.
WNOHANG	Return immediately.
WNOWAIT	Keep the process in a waitable state.
WSTOPPED	Wait for and return the process status of any child that has stopped upon receipt of a signal.
WTRAPPED	Wait for traced process(es) to become trapped or reach a breakpoint (see **ptrace**(2)).

infop must point to a **siginfo_t** structure, as defined in **siginfo**(5). **siginfo_t** is filled in by the system with the status of the process being waited for.

waitid( ), with *idtype* equal to **P_ALL** and *options* equal to **WEXITED | WTRAPPED**, is equivalent to **wait**(2).

RETURN VALUES

If **waitid( )** returns due to a change of state of one of its children, a value of 0 is returned. Otherwise, a value of –1 is returned and **errno** is set to indicate the error.

ERRORS

waitid( ) fails if one or more of the following is true.

ECHILD	The set of processes specified by *idtype* and *id* does not contain any unwaited-for processes.
EFAULT	*infop* points to an illegal address.

EINTR	**waitid( )** was interrupted due to the receipt of a signal by the calling process.
EINVAL	An invalid value was specified for *options*.
EINVAL	*idtype* and *id* specify an invalid set of processes.

SEE ALSO **intro**(2), **exec**(2), **exit**(2), **fork**(2), **pause**(2), **ptrace**(2), **sigaction**(2), **wait**(2), **signal**(3C), **siginfo**(5)

NAME

waitpid – wait for child process to change state

SYNOPSIS

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

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

waitpid( ) suspends the calling process until one of its children changes state; if a child process changed state prior to the call to **waitpid( )**, return is immediate. *pid* specifies a set of child processes for which status is requested.

If *pid* is equal to **(pid_t)–1**, status is requested for any child process.

If *pid* is greater than **(pid_t)0**, it specifies the process ID of the child process for which status is requested.

If *pid* is equal to **(pid_t)0** status is requested for any child process whose process group ID is equal to that of the calling process.

If *pid* is less than **(pid_t)–1**, status is requested for any child process whose process group ID is equal to the absolute value of *pid*.

If **waitpid( )** returns because the status of a child process is available, then that status may be evaluated with the macros defined by **wstat**(5)**.** If the calling process had specified a non-zero value of *stat_loc*, the status of the child process will be stored in the location pointed to by *stat_loc*.

The *options* argument is constructed from the bitwise inclusive OR of zero or more of the following flags, defined in the header **<sys/wait.h>**:

WCONTINUED The status of any continued child process specified by *pid*, whose status has not been reported since it continued, is also reported to the calling process.

WNOHANG **waitpid( )** will not suspend execution of the calling process if status is not immediately available for one of the child processes specified by *pid*.

WNOWAIT Keep the process whose status is returned in *stat_loc* in a waitable state. The process may be waited for again with identical results.

WUNTRACED The status of any child processes specified by *pid* that are stopped, and whose status has not yet been reported since they stopped, is also reported to the calling process.

waitpid( ) with *options* equal to **0** and *pid* equal to **(pid_t)–1** is identical to a call to **wait**(2).

RETURN VALUES

If **waitpid()** returns because the status of a child process is available, this function returns a value equal to the process ID of the child process for which status is reported. If **waitpid()** returns due to the delivery of a signal to the calling process, a value of –1 is returned and **errno** is set to **EINTR**. If this 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 0 is returned. Otherwise, a value of –1 is returned, and **errno** is set to indicate the error.

ERRORS

waitpid() will fail if one or more of the following is true:

ECHILD The process or process group specified by *pid* does not exist or is not a child of the calling process or can never be in the states specified by *options*.

EINTR **waitpid()** was interrupted due to the receipt of a signal sent by the calling process.

EINVAL An invalid value was specified for *options*.

SEE ALSO

intro(2), **exec**(2), **exit**(2), **fork**(2), **pause**(2), **ptrace**(2), **sigaction**(2), **signal**(3C), **siginfo**(5), **wstat**(5)

NAME

write, pwrite, writev – write on a file

SYNOPSIS

```
#include <unistd.h>

ssize_t write(int fildes, const void *buf, size_t nbyte);

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

ssize_t pwrite(int fildes, const void *buf, size_t nbyte, off_t offset);

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

int writev(int fildes, const struct iovec *iov, int iovcnt);
```

MT-LEVEL

write() is Async-Signal-Safe

DESCRIPTION

write() attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file descriptor specified by *fildes*. If *nbyte* is zero and the file is a regular file, **write()** returns zero and has no other results.

pwrite() performs the same action as **write()**, except that it writes into a given position without changing the file pointer. The first three arguments to **pwrite()** are the same as **write()** with the addition of a fourth argument *offset* for the desired position inside the file.

writev() performs the same action as **write()**, but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], ..., *iov*[*iovcnt* – 1]. The *iovcnt* buffer is valid if greater than 0 and less than or equal to **{IOV_MAX}**. (See **intro**(2) for a definition of **{IOV_MAX}**).

The **iovec** structure contains the following members:

```
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 all data from an area before proceeding to the next.

On devices capable of seeking, the actual writing of data starts at the position in the file indicated by the file pointer. On return from **write()**, the file pointer is incremented by the number of bytes actually written. On a regular file, if the incremented file pointer is greater than the length of the file, the length of the file is set to the new file pointer.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the **O_APPEND** flag of the file status flags is set, the file pointer is set to the end of the file prior to each **write()**. The system guarantees that no intervening file modification operation will occur between changing the file offset and the write operation.

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, **write( )** does not return until the data has been physically updated.

A **write( )** to a regular file is blocked if mandatory file / record locking is set (see **chmod**(2)), and there is a record lock owned by another process on the segment of the file to be written:

- If **O_NDELAY** or **O_NONBLOCK** is set, **write( )** returns **–1** and sets **errno** to **EAGAIN**.
- If **O_NDELAY** and **O_NONBLOCK** are clear, **write( )** sleeps until all blocking locks are removed or the **write( )** is terminated by a signal.

If a **write( )** requests that more bytes be written than there is room for—for example, if the write would exceed the process file size limit (see **getrlimit**(2) and **ulimit**(2)), the system file size limit, or the free space on the device—only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A **write( )** of 512-bytes returns 20. The next **write( )** of a non-zero number of bytes gives a failure return (except as noted for pipes and FIFO below).

Write requests to a pipe or FIFO are handled the same as a regular file with the following exceptions:

- There is no file offset associated with a pipe, hence each write request appends to the end of the pipe.
- Write requests of **{PIPE_BUF}** bytes or less are guaranteed not to be interleaved with data from other processes doing writes on the same pipe. Writes of greater than **{PIPE_BUF}** bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the **O_NONBLOCK** or **O_NDELAY** flags are set.
- If **O_NONBLOCK** and **O_NDELAY** are clear, a write request may cause the process to block, but on normal completion it returns *nbyte*.
- If **O_NONBLOCK** and **O_NDELAY** are set, **write( )** does not block the process. If a **write( )** request for **{PIPE_BUF}** or fewer bytes succeeds completely **write( )** returns *nbyte*. Otherwise, if **O_NONBLOCK** is set, it returns **–1** and sets **errno** to **EAGAIN** or if **O_NDELAY** is set, it returns **0**. A **write( )** request for greater than **{PIPE_BUF}** bytes transfers what it can and returns the number of bytes written or it transfers no data and, if **O_NONBLOCK** is set, returns **–1** with **errno** set to **EAGAIN** or if **O_NDELAY** is set, it returns **0**. Finally, if a 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.

When attempting to write to a file descriptor (other than a pipe, FIFO, or stream) that supports nonblocking writes and cannot accept the data immediately:

- If **O_NONBLOCK** and **O_NDELAY** are clear, **write( )** blocks until the data can be

accepted.

- If **O_NONBLOCK** or **O_NDELAY** is set, **write()** does not block the process. If some data can be written without blocking the process, **write()** writes what it can and returns the number of bytes written. Otherwise, if **O_NONBLOCK** is set, it returns **–1** and sets **errno** to **EAGAIN** or if **O_NDELAY** is set, it returns **0**.

For STREAMS files (see **intro**(2) and **streamio**(7I)), the operation of **write()** is determined by the values of the minimum and maximum *nbyte* range ("packet size") accepted by the stream. These values are contained in the topmost stream module, and can not be set or tested from user level. If *nbyte* falls within the packet size range, *nbyte* bytes are written. If *nbyte* does not fall within the range and the minimum packet size value is zero, **write()** breaks the buffer into maximum packet size segments prior to sending the data downstream (the last segment may be smaller than the maximum packet size). If *nbyte* does not fall within the range and the minimum value is non-zero, **write()** fails and sets **errno** to **ERANGE**. Writing a zero-length buffer (*nbyte* is zero) to a STREAMS device sends a zero length message with zero returned. However, writing a zero-length buffer to a pipe or FIFO sends no message and zero is returned. The user program may issue the **I_SWROPT ioctl**(2) to enable zero-length messages to be sent across the pipe or FIFO (see **streamio**(7I)).

When writing to a stream, data messages are created with a priority band of zero. When writing to a stream that is not a pipe or FIFO:

- If **O_NDELAY** and **O_NONBLOCK** are not set, and the stream cannot accept data (the stream write queue is full due to internal flow control conditions), **write()** blocks until data can be accepted.
- If **O_NDELAY** or **O_NONBLOCK** is set and the stream cannot accept data, **write()** returns **-1** and sets **errno** to **EAGAIN**.
- If **O_NDELAY** or **O_NONBLOCK** is set and part of the buffer has already been written when a condition occurs in which the stream cannot accept additional data, **write()** terminates and returns the number of bytes written.

RETURN VALUES

On success, **write()** returns the number of bytes actually written. Otherwise, it returns **-1** and sets **errno** to indicate the error.

ERRORS

write(), pwrite(), and **writev()** fail and the file pointer remains unchanged if one or more of the following are true:

EAGAIN Mandatory file/record locking is set, **O_NDELAY** or **O_NONBLOCK** is set, and there is a blocking record lock.

Total amount of system memory available when reading using raw I/O is temporarily insufficient.

An attempt is made to write to a stream that can not accept data with the **O_NDELAY** or **O_NONBLOCK** flag set.

If a write to a pipe or FIFO of **{PIPE_BUF}** bytes or less is requested and less than *nbytes* of free space is available.

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

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

EFAULT *buf* points to an illegal address.

EFBIG An attempt is made to write a file that exceeds the process's file size limit or the maximum file size (see **getrlimit**(2) and **ulimit**(2)).

EINTR A signal was caught during the write operation and no data was transferred.

EINVAL An attempt is made to write to a stream linked below a multiplexor.

EIO The process is in the background and is attempting to write to its controlling terminal whose **TOSTOP** flag is set; the process is neither ignoring nor blocking **SIGTTOU** signals, and the process group of the process is orphaned.

ENOLCK Enforced record locking was enabled and **{LOCK_MAX}** regions are already locked in the system.

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

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

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

ENOSR An attempt is made to write to a stream with insufficient STREAMS memory resources available in the system.

ENXIO A hangup occurred on the stream being written to.

EPIPE and **SIGPIPE** signal

An attempt is made to write to a pipe that is not open for reading by any process (or to a file descriptor created by **socket**(3N), using type **SOCK_STREAM** that is no longer connected to a peer endpoint). 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.

EPIPE An attempt is made to write to a FIFO that is not open for reading by any process.

An attempt is made to write to a pipe that has only one end open.

ERANGE An attempt is made to write to a stream with *nbyte* outside specified minimum and maximum write range, and the minimum value is nonzero.

In addition, **writev( )** may return one of the following errors:

EINVAL *iovcnt* was less than or equal to 0, or greater than {**IOV_MAX**}.

EINVAL One of the **iov_len** values in the *iov* array was negative.

EINVAL The sum of the **iov_len** values in the *iov* array overflowed an **int**.

In addition, **pwrite( )** fails and the file pointer remains unchanged if the following is true:

ESPIPE *fildes* is associated with a pipe or fifo.

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.

Upon successful completion **write( )** and **writev( )** mark for update the **st_ctime** and **st_mtime** fields of the file.

SEE ALSO **chmod**(2), **creat**(2), **dup**(2), **fcntl**(2), **getrlimit**(2), **intro**(2), **ioctl**(2), **lseek**(2), **open**(2), **pipe**(2), **ulimit**(2), **socket**(3N), **streamio**(7I)

NAME

yield – yield execution to another lightweight process

SYNOPSIS

```
#include <unistd.h>
void yield(void);
```

DESCRIPTION

yield( ) causes the current lightweight process to yield its execution in favor of another lightweight process with the same or greater priority.

SEE ALSO

thr_yield(3T)

Index

G

H

I

K

L

M

N

O

R

S

T

U

V

W

Y