man pages section 2: System Calls

Beta

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 819–2241–30
December 2009

091224@23031

Contents

Preface

Both novice users and those familar with the SunOS operating system can use online man pages to obtain information about the system and its features. 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.

Overview

The following contains a brief description of each man page section 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.
- 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.
- 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 9E describes the DDI (Device Driver Interface)/DKI (Driver/Kernel Interface), DDI-only, and DKI-only entry-point routines a developer can 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 path name is shown. 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:

- [ ] — Brackets. The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.
- . . . — Ellipses. Several values can 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 a 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.

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, and functions are described under USAGE.

IOCTL — This section appears on pages in Section 7 only. Only the device class that 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(7I)`.
OPTIONS	This secton 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 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 lists special rules, features, and commands that require in-depth explanations. The subsections listed here 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 superuser, `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 VARIABLES	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 other than zero for various error conditions.
FILES	This section lists all file names 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.
ATTRIBUTES	This section lists characteristics of commands, utilities, and device drivers by defining the attribute type and its corresponding value. See `attributes(5)` for more information.
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.
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.

REFERENCE

Introduction

Name Intro – introduction to system calls and error numbers

Synopsis

```
#include <errno.h>
```

Description A system call is a C library function that requests a service from the system, such as getting the time of day. This request is performed in the kernel. The library interface executes a trap into the kernel, which actually executes the system call code.

Most system calls return one or more error conditions. An error condition is indicated by an otherwise impossible return 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`, which 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 `-mt` option must be specified on the command line at compilation time (see `threads(5)`). When the `-mt` option is specified, `errno` becomes a macro that enables each thread to have its own `errno`. This `errno` macro can be used on either side of the assignment as though it were a variable.

An error value listed as "will fail" describes a condition whose detection and reporting is mandatory for an implementation that conforms to the Single UNIX Specification (SUS). An application can rely on this condition being detected and reported. An error value listed as "may fail" describes a condition whose detection and reporting is optional for an implementation that conforms to the SUS. An application should not rely this condition being detected and reported. An application that relies on such behavior cannot be assured to be portable across conforming implementations. If more than one error occurs in processing a function call, any one of the possible errors might may be returned, as the order of detection is undefined. See `standards(5)` for additional information regarding the Single UNIX Specification.

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	Lacking appropriate privileges Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or an appropriately privileged process. It is also returned for attempts by ordinary users to perform operations allowed only to processes with certain privileges. The manual pages for individual functions document which privileges are needed to override the restriction.
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 function. If execution is resumed after processing the signal, it will appear as if the interrupted function call returned this error condition.
	In a multithreaded application, `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 functions (see `exec(2)`). 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(2)` (respectively, `write(2)`) request is made to a file that is open only for writing (respectively, reading).
10 ECHILD	No child processes

	A wait(3C) function call 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(2) function failed because the system's process table is full or the user is not allowed to create any more processes, or a call failed because of insufficient memory or swap space.
12 ENOMEM	Not enough space During execution of brk() or sbrk() (see brk(2)), or one of the exec family of functions, 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(2) function.
13 EACCES	Permission denied An attempt was made to access a file in a way forbidden by the protection system. The manual pages for individual functions document which privileges are needed to override 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(2) function).
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 mutexes, semaphores, condition variables, and r/w locks, to indicate that a lock is held, and 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(2)` function).
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(2)` function).
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(3C)` or `kill(2)` function, or an unsupported operation related to extended attributes was attempted.
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(2)` function 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`n; the file size exceeds the maximum supported by the file system; or the file size exceeds the offset maximum of the file descriptor. See the `File Descriptor` subsection of the `DEFINITIONS` section below.
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(2)` function, 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(2)` function 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 function
	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 `msgrcv(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 mutexes, 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 EDQUOT	Disc quota exceeded
	A `write(2)` to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because the user's quota of disk blocks was exhausted, or the allocation of an inode for a newly created file failed because the user's quota of inodes was exhausted.
58-59	Reserved
60 ENOSTR	Device not a stream
	A `putmsg(2)` or `getmsg(2)` 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(2)` call has expired. The cause of this error is device-specific and could indicate either a hardware or software failure, or perhaps a timeout value that is too short for the specific operation. The status of the `ioctl()` operation is indeterminate. This is also returned in the case of `_lwp_cond_timedwait(2)` or `cond_timedwait(3C)`.
63 ENOSR	Out of stream resources
	During a STREAMS `open(2)` call, 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.
65 ENOPKG	Package not installed
	This error occurs when users attempt to use a call from a package which has not been installed.
71 EPROTO	Protocol error

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

77 EBADMSG — Not a data message

During a `read(2)`, `getmsg(2)`, or `ioctl(2)` `I_RECVFD` call to a STREAMS device, something has come to the head of the queue that can not be processed. That something depends on the 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.h(3HEAD)`.

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 share library

Trying to `exec` an `a.out` that requires a static shared library and the static shared library does not exist or the user does not 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 *System Administration Guide: IP Services*
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 EMGSIZE	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`(3SOCKET) or `sendmsg`(3SOCKET) 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`(3SOCKET) or `send`(3SOCKET) request failed because the connected party did not properly respond after a period of time; or a `write`(2) or `fsync`(3C) request failed because a file is on an NFS file system mounted with the *soft* option.
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 catalogs the list of files, including directories (sub-directories), 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 for a file are granted to a process if one or more of the following are true:

- 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.
- The read, write, or execute mode bit is not set but the process has the discretionary file access override privilege for the corresponding mode bit: {PRIV_FILE_DAC_READ} for the read bit {PRIV_FILE_DAC_WRITE} for the write bit, {PRIV_FILE_DAC_SEARCH} for the execute bit on directories, and {PRIV_FILE_DAC_EXECUTE} for the executable bit on plain files.

Otherwise, the corresponding permissions are denied.

File Descriptor — A file descriptor is a small integer used to perform 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 calls such as open(2) or pipe(2). The file descriptor is used as an argument by calls such as read(2), write(2), ioctl(2), and close(2).

Each file descriptor has a corresponding offset maximum. For regular files that were opened without setting the O_LARGEFILE flag, the offset maximum is 2 Gbyte – 1 byte (2^{31} –1 bytes). For regular files that were opened with the O_LARGEFILE flag set, the offset maximum is 2^{63} –1 bytes.

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.h(3HEAD)), 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(2) 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_t    msg_cbytes;
ulong_t    msg_qnum;
ulong_t    msg_qbytes;
```

```
pid_t      msg_lspid;
pid_t      msg_lrpid;
time_t     msg_stime;
time_t     msg_rtime;
time_t     msg_ctime;
```

The following are descriptions of the `msqid_ds` structure members:

The `msg_perm` member 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_t  seq;    /* slot usage sequence # */
key_t    key;    /* key */
```

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

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

The `msg_cbytes` member is the current number of bytes on the queue.

The `msg_qnum` member is the number of messages currently on the queue.

The `msg_qbytes` member is the maximum number of bytes allowed on the queue.

The `msg_lspid` member is the process ID of the last process that performed a `msgsnd()` operation.

The `msg_lrpid` member is the process id of the last process that performed a `msgrcv()` operation.

The `msg_stime` member is the time of the last `msgsnd()` operation.

The `msg_rtime` member is the time of the last `msgrcv()` operation.

The `msg_ctime` member is the time of the last `msgctl()` operation that changed a member of the above structure.

Message Operation Permissions

In the `msgctl(2)`, `msgget(2)`, `msgrcv(2)`, and `msgsnd(2)` function 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 for a `msqid` are granted to a process if one or more of the following are true:

- The {`PRIV_IPC_DAC_READ`} or {`PRIV_IPC_DAC_WRITE`} privilege is present in the effective set.
- 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.
- Any group ID in the process credentials from the set (`cr_gid`, `cr_groups`) 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.

Offset Maximum An offset maximum is an attribute of an open file description representing the largest value that can be used as a file offset.

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 non-existent file.

Privileged User Solaris software implements a set of privileges that provide fine-grained control over the actions of processes. The possession of of a certain privilege allows a process to perform a specific set of restricted operations. Prior to the Solaris 10 release, a process running with uid 0 was granted all privileges. See `privileges(5)` for the semantics and the degree of backward compatibility awarded to processes with an effective uid of 0.

Process ID Each process in the system is uniquely identified during its lifetime by a positive integer called a process ID. A process ID cannot 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. There are threads within a process with thread IDs `thread_t` and `LWPID_t`. These threads are not visible 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(3C)`.

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.

Processor Set ID The processors in a system may be divided into subsets, known as processor sets. A process bound to one of these sets will run only on processors in that set, and the processors in the set will normally run only processes that have been bound to the set. Each active processor set is identified by a positive integer. See `pset_create(2)`.

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 Resource Limits

Saved resource limits is an attribute of a process that provides some flexibility in the handling of unrepresentable resource limits, as described in the `exec` family of functions and `setrlimit(2)`.

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 just after 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(2)` 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_t          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 */
```

The following are descriptions of the `semid_ds` structure members:

The `sem_perm` member 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_t   seq;    /* slot usage sequence number */
key_t     key;    /* key */
```

The `sem_nsems` member 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.

The `sem_otime` member is the time of the last `semop(2)` operation.

The sem_ctime member is the time of the last semctl(2) operation that changed a member of the above structure.

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

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

The following are descriptions of the sem structure members:

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

The sempid member is equal to the process ID of the last process that performed a semaphore operation on this semaphore.

The semncnt member is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become greater than its current value.

The semzcnt member 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(2) and semctl(2) function 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 for a semid are granted to a process if one or more of the following are true:

- The {PRIV_IPC_DAC_READ} or {PRIV_IPC_DAC_WRITE} privilege is present in the effective set.
- 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(2)` 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 */
size_t            shm_segsz;    /* size of segment */
struct anon_map   *shm_amp;     /* ptr to region structure */
char              pad[4];       /* for swap compatibility */
pid_t             shm_lpid;     /* pid of last operation */
pid_t             shm_cpid;     /* creator pid */
shmatt_t          shm_nattch;   /* number of current attaches */
ulong_t           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 */
```

The following are descriptions of the `shmid_ds` structure members:

The `shm_perm` member 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_t   seq;    /* slot usage sequence # */
key_t     key;    /* key */
```

The `shm_segsz` member specifies the size of the shared memory segment in bytes.

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

The `shm_lpid` member is the process ID of the last process that performed a `shmat()` or `shmdt()` operation (see shmop(2)).

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

The `shm_atime` member is the time of the last `shmat()` operation (see shmop(2)).

The `shm_dtime` member is the time of the last `shmdt()` operation (see shmop(2)).

The `shm_ctime` member is the time of the last shmctl(2) operation that changed one of the members of the above structure.

Shared Memory Operation Permissions

In the shmctl(2), `shmat()`, and `shmdt()` (see shmop(2)) function 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 for a `shmid` are granted to a process if one or more of the following are true:

- The {`PRIV_IPC_DAC_READ`} or {`PRIV_IPC_DAC_WRITE`} privilege is present in the effective set.
- 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 principal functions of the stream head are processing STREAMS-related system calls and passing data and information between a user process and the stream.

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.

Acknowledgments

Sun Microsystems, Inc. gratefully acknowledges The Open Group for permission to reproduce portions of its copyrighted documentation. Original documentation from The Open Group can be obtained online at `http://www.opengroup.org/bookstore/`.

The Institute of Electrical and Electronics Engineers and The Open Group, have given us permission to reprint portions of their documentation.

In the following statement, the phrase “this text" refers to portions of the system documentation.

Portions of this text are reprinted and reproduced in electronic form in the SunOS Reference Manual, from IEEE Std 1003.1, 2004 Edition, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 6, Copyright (C) 2001-2004 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. In the event of any discrepancy between these versions and the original IEEE and The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at `http://www.opengroup.org/unix/online.html`.

This notice shall appear on any product containing this material.

See Also standards(5), threads(5)

REFERENCE

System Calls

Name access, faccessat – determine accessibility of a file

Synopsis

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

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

int faccessat(int fd, const char *path, int amode, int flag);
```

Description The `access()` function checks the file named by the pathname pointed to by the *path* argument 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 value of *amode* is either the bitwise inclusive OR of the access permissions to be checked (`R_OK`, `W_OK`, `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".

If any access permissions are to be checked, each will be checked individually, as described in `Intro(2)`. If the process has appropriate privileges, an implementation may indicate success for `X_OK` even if none of the execute file permission bits are set.

The `faccessat()` function is equivalent to the `access()` function, except in the case where *path* specifies a relative path. In this case the file whose accessibility is to be determined is located relative to the directory associated with the file descriptor *fd* instead of the current working directory.

If `faccessat()` is passed in the *fd* parameter the special value `AT_FDCWD`, defined in <`fcntl.h`>, the current working directory is used and the behavior is identical to a call to `access()`.

Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in <`fcntl.h`>:

`AT_EACCESS` The checks for accessibility are performed using the effective user and group IDs instead of the real user and group ID as required in a call to `access()`.

Return Values If the requested access is permitted, `access()` and `faccessat()` succeed and return `0`. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `access()` and `faccessat()` functions will fail if:

`EACCES` Permission bits of the file mode do not permit the requested access, or search permission is denied on a component of the path prefix.

`EFAULT` The *path* argument points to an illegal address.

`EINTR` A signal was caught during the `access()` function.

`ELOOP` Too many symbolic links were encountered in resolving *path*, or loop exists in symbolic links encountered during resolution of the *path* argument.

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

`ENOENT` A component of *path* does not name an existing file or *path* is an empty string.

`ENOLINK` The *path* argument 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.

`ENXIO` The *path* argument points to a character or block device special file and the corresponding device has been retired by the fault management framework.

`EROFS` Write access is requested for a file on a read-only file system.

The `faccessat()` function will fail if:

`EBADF` The *path* argument does not specify an absolute path and the *fd* argument is neither `AT_FDCWD` nor a valid file descriptor open for reading or searching.

The `access()` and `faccessat()` functions may fail if:

`EINVAL` The value of the *amode* argument is invalid.

`ENAMETOOLONG` Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {`PATH_MAX`}.

`ETXTBSY` Write access is requested for a pure procedure (shared text) file that is being executed.

The `faccessat()` function may fail if:

`EINVAL` The value of the *flag* argument is not valid.

ENOTDIR The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

Usage Additional values of *amode* other than the set defined in the description might be valid, for example, if a system has extended access controls.

The purpose of the faccessat() function is to enable the checking of the accessibility of files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to access(), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the faccessat() function, it can be guaranteed that the file tested for accessibility is located relative to the desired directory.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See below.

For access(), see standards(5).

See Also Intro(2), chmod(2), stat(2), attributes(5), standards(5)

Name acct – enable or disable process accounting

Synopsis

```
#include <unistd.h>

int acct(const char *path);
```

Description The `acct()` function 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 either an `exit(2)` call or a `signal(3C)`). The effective user ID of the process calling `acct()` must have the appropriate privileges.

The *path* argument points to the pathname of the accounting file, whose file format is described on the `acct.h(3HEAD)` manual page.

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, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `acct()` function will fail if:

`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`	The *path* argument 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* argument 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 {`PRIV_SYS_ACCT`} privilege is not asserted in the effective set of the calling process.
`EROFS`	The named file resides on a read-only file system.

See Also `exit(2)`, `acct.h(3HEAD)`, `signal(3C)`, `privileges(5)`

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, void *aclbufp);

int facl(int fildes, int cmd, int nentries, void *aclbufp);
```

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

The following types are supported for *aclbufp*:

`aclent_t` Used by the UFS file system.

`ace_t` Used by the ZFS and NFSv4 file systems.

The following values for *cmd* are supported:

`SETACL` *nentries* `aclent_t` ACL entries, specified in buffer *aclbufp*, are stored in the file's ACL. All directories in the path name must be searchable.

`GETACL` Buffer *aclbufp* is filled with the file's `aclent_t` 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 `aclent_t` ACL is returned. Read access to the file is not required, but all directories in the path name must be searchable.

`ACE_SETACL` *nentries* `ace_t` ACL entries, specified in buffer *aclbufp*, are stored in the file's ACL. All directories in the path name must be searchable. Write ACL access is required to change the file's ACL.

`ACE_GETACL` Buffer *aclbufp* is filled with the file's `ace_t` ACL entries. Read access to the file is required and all directories in the path name must be searchable.

`ACE_GETACLCNT` The number of entries in the file's `ace_t` ACL is returned. Read access to the file is required and all directories in the path name must be searchable.

Return Values Upon successful completion, `acl()` and `facl()` return `0` if *cmd* is `SETACL` or `ACE_SETACL`. If *cmd* is `GETACL`, `GETACLCNT`, `ACE_GETACL` or `ACE_GETACLCNT`, the number of ACL entries is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `acl()` function will fail if:

`EACCES` The caller does not have access to a component of the pathname.

`EFAULT` The *pathp* or *aclbufp* argument points to an illegal address.

EINVAL	The *cmd* argument is not GETACL, SETACL, ACE_GETACL, GETACLCNT, or ACE_GETACLCNT; the *cmd* argument is SETACL and *nentries* is less than 3; or the *cmd* argument is SETACL or ACE_SETACL and the ACL specified in *aclbufp* is not valid.
EIO	A disk I/O error has occurred while storing or retrieving the ACL.
ENOENT	A component of the path does not exist.
ENOSPC	The *cmd* argument is GETACL and *nentries* is less than the number of entries in the file's ACL, or the *cmd* argument is SETACL and there is insufficient space in the file system to store the ACL.
ENOSYS	The *cmd* argument is SETACL or ACE_SETACL and the file specified by *pathp* resides on a file system that does not support ACLs, or the acl() function is not supported by this implementation.
ENOTDIR	A component of the path specified by *pathp* is not a directory, or the *cmd* argument is SETACL or ACE_SETACL and an attempt is made to set a default ACL on a file type other than a directory.
ENOTSUP	The *cmd* argument is GETACL, but the ACL is composed of ace_t entries, and the ACL cannot be translated into aclent_t form. The *cmd* argument is ACE_SETACL, but the underlying filesystem only supports ACLs composed of aclent_t entries and the ACL could not be translated into aclent_t form.
EPERM	The effective user ID does not match the owner of the file and the process does not have appropriate privilege.
EROFS	The *cmd* argument is SETACL or ACE_SETACL and the file specified by *pathp* resides on a file system that is mounted read-only.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

See Also getfacl(1), setfacl(1), aclcheck(3SEC), aclsort(3SEC)

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 The `adjtime()` function 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. 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 *delta* is 0, then *olddelta* returns the status of the effects of the previous `adjtime()` call with no effect on the time correction as a result of this call. 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 a processes with appropriate privileges can adjust the time of day.

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

Return Values Upon successful completion, `adjtime()` returns `0`. Otherwise, it returns `−1` and sets `errno` to indicate the error.

Errors The `adjtime()` function will fail if:

`EFAULT` The *delta* or *olddelta* argument points outside the process's allocated address space, or *olddelta* points to a region of the process's allocated address space that is not writable.

`EINVAL` The `tv_usec` member of *delta* is not within valid range (−1000000 to 1000000).

`EPERM` The {`PRIV_SYS_TIME`} privilege is not asserted in the effective set of the calling process.

Additionally, the `adjtime()` function will fail for 32-bit interfaces if:

`EOVERFLOW` The size of the `tv_sec` member of the `timeval` structure pointed to by *olddelta* is too small to contain the correct number of seconds.

See Also date(1), gettimeofday(3C), privileges(5)

Name alarm – schedule an alarm signal

Synopsis

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

Description The `alarm()` function causes the system to generate a `SIGALRM` signal for the process after the number of real-time seconds specified by seconds have elapsed (see `signal.h`(3HEAD)). Processor scheduling delays may prevent the process from handling the signal as soon as it is generated.

If *seconds* is 0, a pending alarm request, if any, is cancelled. If *seconds* is greater than `LONG_MAX`/*hz*, *seconds* is rounded down to `LONG_MAX`/*hz*. The value of *hz* is normally 100.

Alarm requests are not stacked; only one `SIGALRM` generation can be scheduled in this manner; if the `SIGALRM` signal has not yet been generated, the call will result in rescheduling the time at which the `SIGALRM` signal will be generated.

The `fork`(2) function clears pending alarms in the child process. A new process image created by one of the `exec`(2) functions inherits the time left to an alarm signal in the old process's image.

Return Values If there is a previous alarm request with time remaining, `alarm()` returns a non-zero value that is the number of seconds until the previous request would have generated a `SIGALRM` signal. Otherwise, `alarm()` returns 0.

Errors The `alarm()` function is always successful; no return value is reserved to indicate an error.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `exec`(2), `fork`(2), `signal.h`(3HEAD), `attributes`(5), `standards`(5)

Name audit – write a record to the audit log

Synopsis

```
cc [ flag ... ] file ... -lbsm  -lsocket   -lnsl   [ library... ]
#include <sys/param.h>
#include <bsm/libbsm.h>

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

Description The `audit()` function queues a record for writing to the system audit log. The data pointed to by *record* is queued for 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.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `audit()` function will fail if:

`E2BIG` The record length is greater than the maximum allowed record length.

`EFAULT` The *record* argument points outside the process's allocated address space.

`EINVAL` The header token in the record is invalid.

`ENOTSUP` Solaris Audit is not defined for this system.

`EPERM` The {`PRIV_PROC_AUDIT`} privilege is not asserted in the effective set of the calling process.

Usage Only privileged processes can successfully execute this call.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also `bsmconv(1M)`, `audit(1M)`, `auditd(1M)`, `svcadm(1M)`, `auditon(2)`, `getaudit(2)`, `audit.log(4)`, `attributes(5)`, `privileges(5)`

Notes The functionality described in this man page is available only if the Solaris Auditing has been enabled and the audit daemon `auditd(1M)` has not been disabled by `audit(1M)` or `svcadm(1M)`. See `bsmconv(1M)` for more information.

Name auditon – manipulate auditing

Synopsis

```
cc [ flag... ] file... -lbsm  -lsocket   -lnsl  [ library... ]
#include <sys/param.h>
#include <bsm/libbsm.h>

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

Description The `auditon()` function 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`
: Return the system audit on/off/disabled condition in the integer pointed to by *data*. The following values can be returned:

  `AUC_AUDITING` Auditing has been turned on.

  `AUC_DISABLED` Auditing system has not been enabled.

  `AUC_NOAUDIT` Auditing has been turned off.

  `AUC_NOSPACE` Auditing has blocked due to lack of space in audit partition.

`A_SETCOND`
: Set the system's audit on/off condition to the value in the integer pointed to by *data*. The Solaris Audit subsystem must be enabled by `bsmconv(1M)` before auditing can be turned on. The following audit states can be set:

  `AUC_AUDITING` Turns on audit record generation.

  `AUC_NOAUDIT` Turns off audit record generation.

`A_GETCLASS`
: Return 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`
: Set 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`
: Return the kernel preselection mask in the `au_mask` structure pointed to by *data*. This is the mask used to preselect non-attributable audit events.

`A_SETKMASK`
: Set the kernel preselection mask. The *data* argument points to the `au_mask` structure containing the class mask. This is the mask used to preselect non-attributable audit events.

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

Note that A_GETPINFO can fail if the termial ID contains a network address longer than 32 bits. In this case, the A_GETPINFO_ADDR command should be used.

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

A_SETPMASK
Set the preselection mask of the specified process. The *data* argument points to the auditpinfo structure containing the process ID and the preselection mask. The other fields of the structure are ignored and should be set to NULL.

A_SETUMASK
Set 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. The other fields of the structure are ignored and should be set to NULL.

A_SETSMASK
Set 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. The other fields of the structure are ignored and should be set to NULL.

A_GETQCTRL
Return 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. The high water mark is the maximum allowed number of undelivered audit records. The low water mark determines when threads blocked on the queue are wakened. Another parameter controls the size of the data buffer used to write data to the audit trail. There is also a parameter that specifies a maximum delay before data is attempted to be written to the audit trail. The audit queue parameters are returned in the au_qctrl structure pointed to by *data*.

A_SETQCTRL
Set the kernel audit queue control parameters as described above in the A_GETQCTRL command. The *data* argument points to the au_qctrl structure containing the audit queue control parameters. The default and maximum values 'A/B' for the audit queue control parameters are:

high water	100/10000 (audit records)
low water	10/1024 (audit records)
output buffer size	1024/1048576 (bytes)
delay	20/20000 (hundredths second)

A_GETCWD
: Return 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 is the length of the buffer.

A_GETCAR
: Return the current active root as kept by the audit subsystem. This path can 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 is the length of the buffer.

A_GETSTAT
: Return the system audit statistics in the `audit_stat` structure pointed to by *data*.

A_SETSTAT
: Reset system audit statistics values. The kernel statistics value is reset if the corresponding field in the statistics structure pointed to by the *data* argument is `CLEAR_VAL`. Otherwise, the value is not changed.

A_GETPOLICY
: Return the audit policy flags in the `uint32_t` pointed to by *data*.

A_SETPOLICY
: Set the audit policy flags to the values in the `uint32_t` 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_AHLT
  : Halt the machine when a non-attributable audit record can not be delivered. The default action is to count the number of events that could not be recorded.

  AUDIT_ARGV
  : Include in the audit record the argument list for a member of the exec(2) family of functions. The default action is not to include this information.

  AUDIT_ARGE
  : Include the environment variables for the execv(2) function 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. The default action is to include only the primary path from the system call.

AUDIT_WINDATA_DOWN
: Include in an audit record any downgraded data moved between windows. This policy is available only if the system is configured with Trusted Extensions. By default, this information is not included.

AUDIT_WINDATA_UP
: Include in an audit record any upgraded data moved between windows. This policy is available only if the system is configured with Trusted Extensions. By default, this information is not included.

AUDIT_PERZONE
: Enable auditing for each local zone. If not set, audit records from all zones are collected in a single log accessible in the global zone and certain auditconfig(1M) operations are disallowed. This policy can be set only from the global zone.

AUDIT_ZONENAME
: Generate a zone ID token with each audit record.

Return Values Upon successful completion, auditon() returns 0. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The auditon() function will fail if:

E2BIG
: The *length* field for the command was too small to hold the returned value.

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

EINVAL
: One of the arguments was illegal, Solaris Audit has not been installed, or the operation is not valid from a local zone.

EPERM
: The {PRIV_SYS_AUDIT} privilege is not asserted in the effective set of the calling process.

  Neither the {PRIV_PROC_AUDIT} nor the {PRIV_SYS_AUDIT} privilege is asserted in the effective set of the calling process and the command is one of A_GETCAR, A_GETCLASS, A_GETCOND, A_GETCWD, A_GETPINFO, A_GETPOLICY.

Usage The auditon() function can be invoked only by processes with appropriate privileges.

The use of auditon() to change system audit state is permitted only in the global zone. From any other zone auditon() returns −1 with errno set to EPERM. The following auditon() commands are permitted only in the global zone: A_SETCOND, A_SETCLASS, A_SETKMASK, A_SETQCTRL, A_SETSTAT, A_SETFSIZE, and A_SETPOLICY. All other auditon() commands are valid from any zone.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	MT-Safe

See Also auditconfig(1M), auditd(1M), bsmconv(1M), audit(2), exec(2), audit.log(4), attributes(5), privileges(5)

Notes The functionality described in this man page is available only if the Solaris Auditing has been enabled. See bsmconv(1M) for more information.

The auditon options that modify or display process-based information are not affected by the "perzone" audit policy. Those that modify system audit data such as the terminal ID and audit queue parameters are valid only in the global zone unless the "perzone" policy is set. The "get" options for system audit data reflect the local zone if "perzone" is set; otherwise they reflects the settings of the global zone.

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(intptr_t incr);
```

Description The `brk()` and `sbrk()` functions 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 is not possible to set the break beyond the `rlim_max` value returned from a call to `getrlimit()`, that is to say, "`end` + `rlim.rlim_max`." See `end(3C)`.

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

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

Return Values Upon successful completion, `brk()` returns `0`. Otherwise, it returns `−1` and sets `errno` to indicate the error.

Upon successful completion, `sbrk()` returns the prior break value. Otherwise, it returns `(void *)−1` and sets `errno` to indicate the error.

Errors The `brk()` and `sbrk()` functions will fail and no additional memory will be allocated if:

`ENOMEM` The data segment size limit as set by `setrlimit()` (see `getrlimit(2)`) would be exceeded; the maximum possible size of a data segment (compiled into the system) would be exceeded; insufficient space exists in the swap area to support the expansion; or 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)`).

Usage The behavior of `brk()` and `sbrk()` is unspecified if an application also uses any other memory functions (such as `malloc(3C)`, `mmap(2)`, `free(3C)`). The `brk()` and `sbrk()` functions have been used in specialized cases where no other memory allocation function provided the same capability. The use of `mmap(2)` is now preferred because it can be used portably with all other memory allocation functions and with any function that uses other allocation functions.

It is unspecified whether the pointer returned by `sbrk()` is aligned suitably for any purpose.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also `exec(2)`, `getrlimit(2)`, `mmap(2)`, `shmop(2)`, `ulimit(2)`, `end(3C)`, `free(3C)`, `malloc(3C)`

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 boundaries.

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);
```

Description The `chdir()` and `fchdir()` functions 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 `/` (slash). The *path* argument points to the path name of a directory. The *fildes* argument is an open file descriptor of a directory.

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

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned, the current working directory is unchanged, and `errno` is set to indicate the error.

Errors The `chdir()` function will fail if:

`EACCES`	Search permission is denied for any component of the path name.
`EFAULT`	The *path* argument 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`	The *path* argument 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.

The `fchdir()` function will fail if:

`EACCES`	Search permission is denied for *fildes*.
`EBADF`	The *fildes* argument 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`	The *fildes* argument 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.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also chroot(2), attributes(5), standards(5)

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);
```

Description The `chmod()` and `fchmod()` functions 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	Sticky bit.
`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 the bitwise OR operation of 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 regular file is not executable and has `S_ISVTX` set, the file is assumed to be a swap file. In this case, the system's page cache will not be used to hold the file's data. If the `S_ISVTX` bit is set on any other file, the results are unspecified.

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. 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, possibly affecting future calls to `open(2)`, `creat(2)`, `read(2)`, and `write(2)` on this file.

If *fildes* references a shared memory object, `fchmod()` need only affect the `S_IRUSR`, `S_IRGRP`, `S_IROTH`, `S_IWUSR`, `S_IWGRP`, `S_IWOTH`, `S_IXUSR`, `S_IXGRP`, and `S_IXOTH` file permission bits.

If *fildes* refers to a socket, `fchmod()` does not fail but no action is taken.

If *fildes* refers to a stream that is attached to an object in the file system name space with `fattach(3C)`, the `fchmod()` call performs no action and returns successfully.

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

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned, the file mode is unchanged, and `errno` is set to indicate the error.

Errors The `chmod()` and `fchmod()` functions will fail if:

`EIO` An I/O error occurred while reading from or writing to the file system.

`EPERM` The effective user ID does not match the owner of the file and the process does not have appropriate privilege.

The {`PRIV_FILE_OWNER`} privilege overrides constraints on ownership when changing permissions on a file.

The {PRIV_FILE_SETID} privilege overrides constraints on ownership when adding the setuid or setgid bits to an executable file or a directory. When adding the setuid bit to a root owned executable, additional restrictions apply. See privileges(5).

The chmod() function will fail if:

EACCES — Search permission is denied on a component of the path prefix of *path*. The privilege {FILE_DAC_SEARCH} overrides file permissions restrictions in that case.

EFAULT — The *path* argument points to an illegal address.

ELOOP — A loop exists in symbolic links encountered during the resolution of the *path* argument.

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 — The *fildes* argument 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.

EROFS — The file referred to by *path* resides on a read-only file system.

The fchmod() function will fail if:

EBADF — The *fildes* argument is not an open file descriptor

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

EROFS — The file referred to by *fildes* resides on a read-only file system.

The chmod() and fchmod() functions may fail if:

EINTR — A signal was caught during execution of the function.

EINVAL — The value of the *mode* argument is invalid.

The chmod() function may fail if:

ELOOP — More than {SYMLOOP_MAX} symbolic links were encountered during the resolution of the *path* argument.

ENAMETOOLONG — As a result of encountering a symbolic link in resolution of the*path* argument, the length of the substituted pathname strings exceeds {PATH_MAX}.

The `fchmod()` function may fail if:

`EINVAL` The *fildes* argument refers to a pipe and the system disallows execution of this function on a pipe.

Examples

EXAMPLE 1 Set Read Permissions for User, Group, and Others

The following example sets read permissions for the owner, group, and others.

```
#include <sys/stat.h>
const char *path;
...
chmod(path, S_IRUSR|S_IRGRP|S_IROTH);
```

EXAMPLE 2 Set Read, Write, and Execute Permissions for the Owner Only

The following example sets read, write, and execute permissions for the owner, and no permissions for group and others.

```
#include <sys/stat.h>
const char *path;
...
chmod(path, S_IRWXU);
```

EXAMPLE 3 Set Different Permissions for Owner, Group, and Other

The following example sets owner permissions for CHANGEFILE to read, write, and execute, group permissions to read and execute, and other permissions to read.

```
#include <sys/stat.h>
#define CHANGEFILE "/etc/myfile"
...
chmod(CHANGEFILE, S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH);
```

EXAMPLE 4 Set and Checking File Permissions

The following example sets the file permission bits for a file named `/home/cnd/mod1`, then calls the `stat(2)` function to verify the permissions.

```
#include <sys/types.h>
#include <sys/stat.h>
int status;
struct stat buffer
...
chmod("home/cnd/mod1", S_IRWXU|S_IRWXG|S_IROTH|S_IWOTH);
status = stat("home/cnd/mod1", &buffer;);
```

Usage If `chmod()` or `fchmod()` is used to change the file group owner permissions on a file with non-trivial ACL entries, only the ACL mask is set to the new permissions and the group owner permission bits in the file's mode field (defined in `mknod(2)`) are unchanged. A non-trivial ACL entry is one whose meaning cannot be represented in the file's mode field alone. The new ACL mask permissions might change the effective permissions for additional users and groups that have ACL entries on the file.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `chmod(1)`, `chown(2)`, `creat(2)`, `fcntl(2)`, `mknod(2)`, `open(2)`, `read(2)`, `rename(2)`, `stat(2)`, `write(2)`, `fattach(3C)`, `mkfifo(3C)`, `stat.h(3HEAD)`, `attributes(5)`, `privileges(5)`, `standards(5)`

Programming Interfaces Guide

Name chown, lchown, fchown, fchownat – 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);

int fchownat(int fildes, const char *path, uid_t owner,
     gid_t group, int flag);
```

Description The `chown()` function 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 `lchown()` function sets the owner ID and group ID of the named file in the same manner as `chown()`, unless 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.

The `fchownat()` function sets the owner ID and group ID of the named file in the same manner as `chown()`. If, however, the *path* argument is relative, the path is resolved relative to the *fildes* argument rather than the current working directory. If the *fildes* argument has the special value `AT_FDCWD`, the path resolution reverts back to current working directory relative. If the *flag* argument is set to `SYMLNK`, the function behaves like `lchown()` with respect to symbolic links. If the *path* argument is absolute, the *fildes* argument is ignored. If the *path* argument is a null pointer, the function behaves like `fchown()`.

If `chown()`, `lchown()`, `fchown()`, or `fchownat()` is invoked by a process that does not have {`PRIV_FILE_SETID`} asserted in its effective set, the set-user-ID and set-group-ID bits of the file mode, `S_ISUID` and `S_ISGID` respectively, are cleared (see `chmod(2)`). Additional restrictions apply when changing the ownership to uid 0.

The operating system defines several privileges to override restrictions on the `chown()` family of functions. When the {`PRIV_FILE_CHOWN`} privilege is asserted in the effective set of the current process, there are no restrictions except in the special circumstances of changing ownership to or from uid 0. When the {`PRIV_FILE_CHOWN_SELF`} privilege is asserted, ownership changes are restricted to the files of which the ownership matches the effective user ID of the current process. If neither privilege is asserted in the effective set of the calling process, ownership changes are limited to changes of the group of the file to the list of supplementary group IDs and the effective group ID.

The operating system provides a configuration option, {`_POSIX_CHOWN_RESTRICTED`}, to control the default behavior of processes and the behavior of the NFS server. If {`_POSIX_CHOWN_RESTRICTED`} is not in effect, the privilege {`PRIV_FILE_CHOWN_SELF`} is

asserted in the inheritable set of all processes unless overridden by policy.conf(4) or user_attr(4). To set this configuration option, include the following line in /etc/system:

set rstchown = 1

To disable this option, include the following line in /etc/system:

set rstchown = 0

See system(4) and fpathconf(2).

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

Return Values Upon successful completion, 0 is returned. Otherwise, −1 is returned, the owner and group of the named file remain unchanged, and errno is set to indicate the error.

Errors All of these functions will fail if:

EPERM The effective user ID does not match the owner of the file and the {PRIV_FILE_CHOWN} privilege is not asserted in the effective set of the calling process, or the {PRIV_FILE_CHOWN_SELF} privilege is not asserted in the effective set of the calling process.

The chown(), lchown(), and fchownat() functions will fail if:

EACCES Search permission is denied on a component of the path prefix of *path*.

EFAULT The *path* argument points to an illegal address and for fchownat(), the file descriptor has the value AT_FDCWD.

EINTR A signal was caught during the execution of the chown() or lchown() function.

EINVAL The *group* or *owner* argument 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*.

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 The *path* argument 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, or the path supplied to fchownat() is relative and the file descriptor provided does not refer to a valid directory.

EROFS The named file resides on a read-only file system.

The `fchown()` and `fchownat()` functions will fail if:

EBADF For `fchown()` the *fildes* argument is not an open file descriptor and.

For `fchownat()`, the *path* argument is not absolute and the *fildes* argument is not `AT_FDCWD` or 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 The *fildes* argument points to a remote machine and the link to that machine is no longer active.

EINVAL The *group* or *owner* argument is out of range.

EROFS The named file referred to by *fildes* resides on a read-only file system.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	See below.
MT-Level	See below.

The `chown()`, `fchown()`, and `lchown()` functions are Standard. The `fchownat()` function is Evolving.

The `chown()` and `fchownat()` functions are Async-Signal-Safe.

See Also

`chgrp(1)`, `chown(1)`, `chmod(2)`, `fpathconf(2)`, `system(4)`, `attributes(5)`, `standards(5)`

Name chroot, fchroot – change root directory

Synopsis

```
#include <unistd.h>

int chroot(const char *path);

int fchroot(int fildes);
```

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

The *path* argument 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 privilege {`PRIV_PROC_CHROOT`} must be asserted in the effective set of the process to change the root directory. While it is always possible to change to the system root using the `fchroot()` function, it is not guaranteed to succeed in any other case, even if *fildes* is valid in all respects.

The ". ." entry in the root directory is interpreted to mean the root directory itself. Therefore, ". ." cannot be used to access files outside the subtree rooted at the root directory. Instead, `fchroot()` can be used to reset the root to a directory that was opened before the root directory was changed.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned, the root directory remains unchanged, and `errno` is set to indicate the error.

Errors The `chroot()` function will fail if:

`EACCES`	Search permission is denied for a component of the path prefix of *dirname*, or search permission is denied for the directory referred to by *dirname*.
`EBADF`	The descriptor is not valid.
`EFAULT`	The *path* argument points to an illegal address.
`EINVAL`	The `fchroot()` function attempted to change to a directory the is not the system root and external circumstances do not allow this.
`EINTR`	A signal was caught during the execution of 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*.
`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`	The *path* argument 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 {PRIV_PROC_CHROOT} privilege is not asserted in the effective set of the calling process.

See Also chroot(1M), chdir(2), privileges(5)

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);
```

Description The `close()` function deallocates the file descriptor indicated by *fildes*. To deallocate means to make the file descriptor available for return by subsequent calls to `open(2)` or other functions that allocate file descriptors. All outstanding record locks owned by the process on the file associated with the file descriptor will be removed (that is, unlocked).

If `close()` is interrupted by a signal that is to be caught, it will return −1 with `errno` set to `EINTR` and the state of *fildes* is unspecified. If an I/O error occurred while reading from or writing to the file system during `close()`, it returns -1, sets `errno` to `EIO`, and the state of *fildes* is unspecified.

When all file descriptors associated with a pipe or FIFO special file are closed, any data remaining in the pipe or FIFO will be discarded.

When all file descriptors associated with an open file description have been closed the open file description will be freed.

If the link count of the file is 0, when all file descriptors associated with the file are closed, the space occupied by the file will be freed and the file will no longer be accessible.

If a streams-based (see `Intro(2)`) *fildes* is closed and the calling process was 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_NONBLOCK` and `O_NDELAY` are not set and there have been no signals posted for the stream, and if there is 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 via an `I_SETCLTIME` `ioctl(2)` request (see `streamio(7I)`). If the `O_NONBLOCK` or `O_NDELAY` flag is set, or if there are any pending signals, `close()` does not wait for output to drain, and dismantles the stream immediately.

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 by `fattach(3C)`, then the last `close()` forces the named end to be detached by `fdetach(3C)`. If the named end has no open file descriptors associated with it and gets detached, the stream associated with that end is also dismantled.

If *fildes* refers to the master side of a pseudo-terminal, a `SIGHUP` signal is sent to the session leader, if any, for which the slave side of the pseudo-terminal is the controlling terminal. It is unspecified whether closing the master side of the pseudo-terminal flushes all queued input and output.

If *fildes* refers to the slave side of a streams-based pseudo-terminal, a zero-length message may be sent to the master.

When there is an outstanding cancelable asynchronous I/O operation against *fildes* when `close()` is called, that I/O operation is canceled. An I/O operation that is not canceled completes as if the `close()` operation had not yet occurred. All operations that are not canceled will complete as if the `close()` blocked until the operations completed.

If a shared memory object or a memory mapped file remains referenced at the last close (that is, a process has it mapped), then the entire contents of the memory object will persist until the memory object becomes unreferenced. If this is the last close of a shared memory object or a memory mapped file and the close results in the memory object becoming unreferenced, and the memory object has been unlinked, then the memory object will be removed.

If *fildes* refers to a socket, `close()` causes the socket to be destroyed. If the socket is connection-mode, and the `SO_LINGER` option is set for the socket with non-zero linger time, and the socket has untransmitted data, then `close()` will block for up to the current linger interval until all data is transmitted.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `close()` function will fail if:

`EBADF` The *fildes* argument is not a valid file descriptor.

`EINTR` The `close()` function was interrupted by a signal.

`ENOLINK` The *fildes* argument is on a remote machine and the link to that machine is no longer active.

`ENOSPC` There was no free space remaining on the device containing the file.

The `close()` function may fail if:

`EIO` An I/O error occurred while reading from or writing to the file system.

Examples **EXAMPLE 1** Reassign a file descriptor.

The following example closes the file descriptor associated with standard output for the current process, re-assigns standard output to a new file descriptor, and closes the original file descriptor to clean up. This example assumes that the file descriptor 0, which is the descriptor for standard input, is not closed.

```
#include <unistd.h>
...
int pfd;
...
close(1);
```

EXAMPLE 1 Reassign a file descriptor. *(Continued)*

```
dup(pfd);
close(pfd);
...
```

Incidentally, this is exactly what could be achieved using:

```
dup2(pfd, 1);
close(pfd);
```

EXAMPLE 2 Close a file descriptor.

In the following example, `close()` is used to close a file descriptor after an unsuccessful attempt is made to associate that file descriptor with a stream.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define LOCKFILE "/etc/ptmp"
...
int pfd;
FILE *fpfd;
...
if ((fpfd = fdopen (pfd, "w")) == NULL) {
        close(pfd);
        unlink(LOCKFILE);
        exit(1);
}
...
```

Usage An application that used the `stdio` function `fopen(3C)` to open a file should use the corresponding `fclose(3C)` function rather than `close()`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `Intro(2)`, `creat(2)`, `dup(2)`, `exec(2)`, `fcntl(2)`, `ioctl(2)`, `open(2)` `pipe(2)`, `fattach(3C)`, `fclose(3C)`, `fdetach(3C)`, `fopen(3C)`, `signal(3C)`, `signal.h(3HEAD)`, `attributes(5)`, `standards(5)`, `streamio(7I)`

Name creat – create a new file or rewrite an existing one

Synopsis

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

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

Description The function call

creat(path, mode)

is equivalent to:

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

Return Values Refer to open(2).

Errors Refer to open(2).

Examples **EXAMPLE 1** Creating a File

The following example creates the file `/tmp/file` with read and write permissions for the file owner and read permission for group and others. The resulting file descriptor is assigned to the *fd* variable.

```
#include <fcntl.h>
...
int fd;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
char *filename = "/tmp/file";
...
fd = creat(filename, mode);
...
```

Usage The `creat()` function has a transitional interface for 64-bit file offsets. See `lf64`(5).

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also open(2), `attributes`(5), `largefile`(5), `lf64`(5), `standards`(5)

Name dup – duplicate an open file descriptor

Synopsis

```
#include <unistd.h>

int dup(int fildes);
```

Description The `dup()` function 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*`)` function call is equivalent to:

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

Return Values Upon successful completion, a non-negative integer representing the file descriptor is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `dup()` function will fail if:

`EBADF` The *fildes* argument is not a valid open file descriptor.

`EINTR` A signal was caught during the execution of the `dup()` function.

`EMFILE` The process has too many open files (see `getrlimit(2)`).

`ENOLINK` The *fildes* argument is on a remote machine and the link to that machine is no longer active.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `close(2)`, `creat(2)`, `exec(2)`, `fcntl(2)`, `getrlimit(2)`, `open(2)`, `pipe(2)`, `dup2(3C)`, `lockf(3C)`, `attributes(5)`, `standards(5)`

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

Synopsis

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ...
     /* const char *argn, (char *)0 */);

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

int execle(const char *path, const char *arg0, ...
     /* const char *argn, (char *)0,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 *)0 */);

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

Description Each of the functions in the exec family replaces the current process image with a new process image. The new image is constructed from a regular, executable file called the *new process image file*. This file is either an executable object file or a file of data for an interpreter. There is no return from a successful call to one of these functions 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 executed, the system invokes 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. The interpreter named by *pathname* must not be an interpreter file.

When a C-language program is executed as a result of this call, it is entered as a C-language function call as follows:

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

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

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

is initialized as a pointer to an array of character pointers to the environment strings. The *argv* and *environ* arrays are each terminated by a null pointer. The null pointer terminating the *argv* array is not counted in *argc*.

The value of *argc* is non-negative, and if greater than 0, *argv*[0] points to a string containing the name of the file. If *argc* is 0, *argv*[0] is a null pointer, in which case there are no arguments. Applications should verify that *argc* is greater than 0 or that *argv*[0] is not a null pointer before dereferencing *argv*[0].

The arguments specified by a program with one of the `exec` functions are passed on to the new process image in the `main()` arguments.

The *path* argument points to a path name that identifies the new process image file.

The *file* argument is used to construct a pathname that identifies the new process image file. If the *file* argument contains a slash character, it is used as the pathname for this file. Otherwise, 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 process image file is not a valid executable object file, `execlp()` and `execvp()` use the contents of that file as standard input to the shell. In this case, the shell becomes the new process image. The standard to which the caller conforms determines which shell is used. See `standards(5)`.

The arguments represented by *arg0*... are pointers to null-terminated character strings. These strings constitute the argument list available to the new process image. The list is terminated by a null pointer. The *arg0* argument should point to a filename that is associated with the process being started by one of the `exec` functions.

The *argv* argument is an array of character pointers to null-terminated strings. The last member of this array must be a null pointer. These strings constitute the argument list available to the new process image. The value in *argv*[0] should point to a filename that is associated with the process being started by one of the `exec` functions.

The *envp* argument is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The *envp* array is terminated by a null pointer. For `execl()`, `execv()`, `execvp()`, and `execlp()`, the C-language 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 image.

The number of bytes available for the new process's combined argument and environment lists is `ARG_MAX`. It is implementation-dependent whether null terminators, pointers, and/or any alignment bytes are included in this total.

File descriptors open in the calling process image remain open in the new process image, except for those whose close-on-exec flag `FD_CLOEXEC` is set; see `fcntl(2)`. For those file descriptors that remain open, all attributes of the open file description, including file locks, remain unchanged.

The preferred hardware address translation size (see `memcntl(2)`) for the stack and heap of the new process image are set to the default system page size.

Directory streams open in the calling process image are closed in the new process image.

The state of conversion descriptors and message catalogue descriptors in the new process image is undefined. For the new process, the equivalent of:

```
setlocale(LC_ALL, "C")
```

is executed at startup.

Signals set to the default action (SIG_DFL) in the calling process image are set to the default action in the new process image (see signal(3C)). Signals set to be ignored (SIG_IGN) by the calling process image are set to be ignored by the new process image. Signals set to be caught by the calling process image are set to the default action in the new process image (see signal.h(3HEAD)). After a successful call to any of the exec functions, alternate signal stacks are not preserved and the SA_ONSTACK flag is cleared for all signals.

After a successful call to any of the exec functions, any functions previously registered by atexit(3C) are no longer registered.

The saved resource limits in the new process image are set to be a copy of the process's corresponding hard and soft resource limits.

If the ST_NOSUID bit is set for the file system containing the new process image file, then the effective user ID and effective group ID are unchanged in the new process image. If the set-user-ID mode bit of the new process image file is set (see chmod(2)), the effective user ID of the new process image is set to the owner ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file. The real user ID and real group ID of the new process image remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image are saved (as the saved set-user-ID and the saved set-group-ID for use by setuid(2).

The privilege sets are changed according to the following rules:

1. The inheritable set, I, is intersected with the limit set, L. This mechanism enforces the limit set for processes.
2. The effective set, E, and the permitted set, P, are made equal to the new inheritable set.

The system attempts to set the privilege-aware state to non-PA both before performing any modifications to the process IDs and privilege sets as well as after completing the transition to new UIDs and privilege sets, following the rules outlined in privileges(5).

If the {PRIV_PROC_OWNER} privilege is asserted in the effective set, the set-user-ID and set-group-ID bits will be honored when the process is being controlled by ptrace(3C). Additional restriction can apply when the traced process has an effective UID of 0. See privileges(5).

Any shared memory segments attached to the calling process image will not be attached to the new process image (see shmop(2)). Any mappings established through mmap() are not preserved across an exec. Memory mappings created in the process are unmapped before the address space is rebuilt for the new process image. See mmap(2).

Memory locks established by the calling process via calls to mlockall(3C) or mlock(3C) are removed. If locked pages in the address space of the calling process are also mapped into the address spaces the locks established by the other processes will be unaffected by the call by this process to the exec function. If the exec function fails, the effect on memory locks is unspecified.

If _XOPEN_REALTIME is defined and has a value other than −1, any named semaphores open in the calling process are closed as if by appropriate calls to sem_close(3C)

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

Timers created by the calling process with timer_create(3C) are deleted before replacing the current process image with the new process image.

For the SCHED_FIFO and SCHED_RR scheduling policies, the policy and priority settings are not changed by a call to an exec function.

All open message queue descriptors in the calling process are closed, as described in mq_close(3C).

Any outstanding asynchronous I/O operations may be cancelled. Those asynchronous I/O operations that are not canceled will complete as if the exec function had not yet occurred, but any associated signal notifications are suppressed. It is unspecified whether the exec function itself blocks awaiting such I/O completion. In no event, however, will the new process image created by the exec function be affected by the presence of outstanding asynchronous I/O operations at the time the exec function is called.

All active contract templates are cleared (see contract(4)).

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

- controlling terminal
- current working directory
- file-locks (see fcntl(2) and lockf(3C))
- file mode creation mask (see umask(2))
- file size limit (see ulimit(2))
- limit privilege set
- nice value (see nice(2))
- parent process ID
- pending signals (see sigpending(2))
- privilege debugging flag (see privileges(5) and getpflags(2))
- process ID

- process contract (see contract(4) and process(4))
- process group ID
- process signal mask (see sigprocmask(2))
- processor bindings (see processor_bind(2))
- processor set bindings (see pset_bind(2))
- project ID
- real group ID
- real user ID
- resource limits (see getrlimit(2))
- root directory
- scheduler class and priority (see priocntl(2))
- semadj values (see semop(2))
- session membership (see exit(2) and signal(3C))
- supplementary group IDs
- task ID
- time left until an alarm clock signal (see alarm(2))
- tms_utime, tms_stime, tms_cutime, and tms_cstime (see times(2))
- trace flag (see ptrace(3C) request 0)

A call to any exec function from a process with more than one thread results in all threads being terminated and the new executable image being loaded and executed. No destructor functions will be called.

Upon successful completion, each of the functions in the exec family marks for update the st_atime field of the file. If an exec function failed but was able to locate the *process image file*, whether the st_atime field is marked for update is unspecified. Should the function succeed, the process image file is considered to have been opened with open(2). The corresponding close(2) is considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the exec functions. The *argv*[ ] and *envp*[ ] arrays of pointers and the strings to which those arrays point will not be modified by a call to one of the exec functions, except as a consequence of replacing the process image.

The saved resource limits in the new process image are set to be a copy of the process's corresponding hard and soft limits.

Return Values If a function in the exec family returns to the calling process image, an error has occurred; the return value is −1 and errno is set to indicate the error.

Errors The exec functions will fail if:

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.

The new process file is not an ordinary file.

The new process file mode denies execute permission.

The {FILE_DAC_SEARCH} privilege overrides the restriction on directory searches.

The {FILE_DAC_EXECUTE} privilege overrides the lack of 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.

EINVAL The new process image file has the appropriate permission and has a recognized executable binary format, but the system does not support execution of a file with this format.

EINTR A signal was caught during the execution of one of the functions in the *exec* family.

ELOOP Too many symbolic links were encountered in translating *path* or *file*.

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.

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

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

The exec functions, except for `execlp()` and `execvp()`, will fail if:

ENOEXEC The new process image file has the appropriate access permission but is not in the proper format.

The exec functions may fail if:

ENAMETOOLONG Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

ENOMEM The new process image requires more memory than is allowed by the hardware or system-imposed by memory management constraints. See brk(2).

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

Usage As the state of conversion descriptors and message catalogue descriptors in the new process image is undefined, portable applications should not rely on their use and should close them prior to calling one of the exec functions.

Applications that require other than the default POSIX locale should call setlocale(3C) with the appropriate parameters to establish the locale of thenew process.

The *environ* array should not be accessed directly by the application.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	See below.
Standard	See standards(5).

The execle() and execve() fucntions are Async-Signal-Safe.

See Also ksh(1), ps(1), sh(1), alarm(2), brk(2), chmod(2), exit(2), fcntl(2), fork(2), getpflags(2), getrlimit(2), memcntl(2), mmap(2), nice(2), priocntl(2), profil(2), semop(2), shmop(2), sigpending(2), sigprocmask(2), times(2), umask(2), lockf(3C), ptrace(3C), setlocale(3C), signal(3C), system(3C), timer_create(3C), a.out(4), contract(4), process(4), attributes(5), environ(5), privileges(5), standards(5)

Warnings If a program is setuid to a user ID other than the superuser, 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, _exit – terminate process

Synopsis

```
#include <stdlib.h>

void exit(int status);

void _Exit(int status);

#include <unistd.h>

void _exit(int status);
```

Description The `exit()` function first calls all functions registered by `atexit(3C)`, in the reverse order of their registration, except that a function is called after any previously registered functions that had already been called at the time it was registered. Each function is called as many times as it was registered. If, during the call to any such function, a call to the `longjmp(3C)` function is made that would terminate the call to the registered function, the behavior is undefined.

If a function registered by a call to `atexit(3C)` fails to return, the remaining registered functions are not called and the rest of the `exit()` processing is not completed. If `exit()` is called more than once, the effects are undefined.

The `exit()` function then flushes all open streams with unwritten buffered data, closes all open streams, and removes all files created by `tmpfile(3C)`.

The `_Exit()` and `_exit()` functions are functionally equivalent. They do not call functions registered with `atexit()`, do not call any registered signal handlers, and do not flush open streams.

The `_exit()`, `_Exit()`, and `exit()` functions terminate the calling process with the following consequences:

- All of the file descriptors, directory streams, conversion descriptors and message catalogue descriptors open in the calling process are closed.
- If the parent process of the calling process is executing a `wait(3C)`, `wait3(3C)`, `waitid(2)`, or `waitpid(3C)`, and has neither set its `SA_NOCLDWAIT` flag nor set `SIGCHLD` to `SIG_IGN`, it is notified of the calling process's termination and the low-order eight bits (that is, bits 0377) of *status* are made available to it. If the parent is not waiting, the child's status will be made available to it when the parent subsequently executes `wait()`, `wait3()`, `waitid()`, or `waitpid()`.
- If the parent process of the calling process is not executing a `wait()`, `wait3()`, `waitid()`, or `waitpid()`, and has not set its `SA_NOCLDWAIT` flag, or set `SIGCHLD` to `SIG_IGN`, the calling process is transformed into a *zombie process*. A *zombie process* is an inactive process and it will be deleted at some later time when its parent process executes `wait()`, `wait3()`, `waitid()`, or `waitpid()`. A zombie process 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(2)` function.

- Termination of a process does not directly terminate its children. The sending of a SIGHUP signal as described below indirectly terminates children in some circumstances.
- A SIGCHLD will be sent to the parent process.
- The parent process ID of all of the calling process's existing child processes and zombie processes is set to 1. That is, these processes are inherited by the initialization process (see Intro(2)).
- Each mapped memory object is unmapped.
- Each attached shared-memory segment is detached and the value of shm_nattch (see shmget(2)) in the data structure associated with its shared memory ID is decremented by 1.
- For each semaphore for which the calling process has set a semadj value (see semop(2)), that value is added to the semval of the specified semaphore.
- If the process is a controlling process, the SIGHUP signal will be sent to each process in the foreground process group of the controlling terminal belonging to the calling process.
- If the process is a controlling process, the controlling terminal associated with the session is disassociated from the session, allowing it to be acquired by a new controlling process.
- If the exit of the process causes a process group to become orphaned, and if any member of the newly-orphaned process group is stopped, then a SIGHUP signal followed by a SIGCONT signal will be sent to each process in the newly-orphaned process group.
- If the parent process has set its SA_NOCLDWAIT flag, or set SIGCHLD to SIG_IGN, the status will be discarded, and the lifetime of the calling process will end immediately.
- 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(3C). All open message queues in the process are closed as if by appropriate calls to mq_close(3C). 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)).
- An extended accounting record is written to the extended process accounting file if the system's extended process accounting facility is enabled (see acctadm(1M)).
- If the current process is the last process within its task and if the system's extended task accounting facility is enabled (see acctadm(1M)), an extended accounting record is written to the extended task accounting file.

Return Values These functions do not return.

Errors No errors are defined.

Usage Normally applications should use `exit()` rather than `_exit()`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	See below.
Standard	See `standards(5)`.

The `_exit()` and `_Exit()` functions are Async-Signal-Safe.

See Also `acctadm(1M)`, `Intro(2)`, `acct(2)`, `close(2)`, `memcntl(2)`, `semop(2)`, `shmget(2)`, `sigaction(2)`, `times(2)`, `waitid(2)`, `atexit(3C)`, `fclose(3C)`, `mq_close(3C)`, `plock(3C)`, `signal.h(3HEAD)`, `tmpfile(3C)`, `wait(3C)`, `wait3(3C)`, `waitpid(3C)`, `attributes(5)`, `standards(5)`

Name fcntl – file control

Synopsis

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

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

Description The `fcntl()` function provides for control over open files. The *fildes* argument is an open file descriptor.

The `fcntl()` function can take a third argument, *arg*, whose data type, value, and use depend upon the value of *cmd*. The *cmd* argument specifies the operation to be performed by `fcntl()`.

The values for *cmd* are defined in <`fcntl.h`> and include:

`F_DUPFD` Return a new file descriptor which is the lowest numbered available (that is, not already open) file descriptor greater than or equal to the third argument, *arg*, taken as an integer of type `int`. The new file descriptor refers to the same open file description as the original file descriptor, and shares any locks. The `FD_CLOEXEC` flag associated with the new file descriptor is cleared to keep the file open across calls to one of the `exec(2)` functions.

`F_DUP2FD` Similar to `F_DUPFD`, but always returns *arg*. `F_DUP2FD` closes *arg* if it is open and not equal to *fildes*. `F_DUP2FD` is equivalent to `dup2`(*fildes*, *arg*).

`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 *arg*. The data type `struct flock` is defined in the <`fcntl.h`> header (see `fcntl.h(3HEAD)`) and is described below. Note that all file systems might not support all possible variations of `F_FREESP` arguments. In particular, many file systems allow space to be freed only at the end of a file.

`F_FREESP64` Equivalent to `F_FREESP`, but takes a `struct flock64` argument rather than a `struct flock` argument.

`F_ALLOCSP` Allocate space for a section of the ordinary file *fildes*. The section is specified by a variable of data type `struct flock` pointed to by *arg*. The data type `struct flock` is defined in the <`fcntl.h`> header (see `fcntl.h(3HEAD)` and is described below.

`F_ALLOCSP64` Equivalent to `F_ALLOCSP`, but takes a `struct flock64` argument rather than a `struct flock` argument.

`F_GETFD` Get the file descriptor flags defined in <`fcntl.h`> that are associated with the file descriptor *fildes*. File descriptor flags are associated with a single file descriptor and do not affect other file descriptors that refer to the same file.

F_GETFL — Get the file status flags and file access modes, defined in <fcntl.h>, for the file descriptor specified by *fildes*. The file access modes can be extracted from the return value using the mask O_ACCMODE, which is defined in <fcntl.h>. File status flags and file access modes do not affect other file descriptors that refer to the same file with different open file descriptions.

F_GETOWN — If *fildes* refers to a socket, get the process or process group ID specified to receive SIGURG signals when out-of-band data is available. Positive values indicate a process ID; negative values, other than −1, indicate a process group ID. If *fildes* does not refer to a socket, the results are unspecified.

F_GETXFL — Get the file status flags, file access modes, and file creation and assignment flags, defined in <fcntl.h>, for the file descriptor specified by *fildes*. The file access modes can be extracted from the return value using the mask O_ACCMODE, which is defined in <fcntl.h>. File status flags, file access modes, and file creation and assignment flags do not affect other file descriptors that refer to the same file with different open file descriptions.

F_SETFD — Set the file descriptor flags defined in <fcntl.h>, that are associated with *fildes*, to the third argument, *arg*, taken as type int. If the FD_CLOEXEC flag in the third argument is 0, the file will remain open across the exec() functions; otherwise the file will be closed upon successful execution of one of the exec() functions.

F_SETFL — Set the file status flags, defined in <fcntl.h>, for the file descriptor specified by *fildes* from the corresponding bits in the *arg* argument, taken as type int. Bits corresponding to the file access mode and file creation and assignment flags that are set in *arg* are ignored. If any bits in *arg* other than those mentioned here are changed by the application, the result is unspecified.

F_SETOWN — If *fildes* refers to a socket, set the process or process group ID specified to receive SIGURG signals when out-of-band data is available, using the value of the third argument, *arg*, taken as type int. Positive values indicate a process ID; negative values, other than −1, indicate a process group ID. If *fildes* does not refer to a socket, the results are unspecified.

The following commands are available for advisory record locking. Record locking is supported for regular files, and may be supported for other files.

F_GETLK — Get the first lock which blocks the lock description pointed to by the third argument, *arg*, taken as a pointer to type struct flock, defined in <fcntl.h>. The information retrieved overwrites the information passed to fcntl() in the structure flock. If no lock is found that would prevent this lock from being created, then the structure will be left unchanged except for the lock type which will be set to F_UNLCK.

F_GETLK64 Equivalent to F_GETLK, but takes a struct flock64 argument rather than a struct flock argument.

F_SETLK Set or clear a file segment lock according to the lock description pointed to by the third argument, *arg*, taken as a pointer to type struct flock, defined in <fcntl.h>. F_SETLK is used to establish shared (or read) locks (F_RDLCK) or exclusive (or write) locks (F_WRLCK), as well as to remove either type of lock (F_UNLCK). F_RDLCK, F_WRLCK and F_UNLCK are defined in <fcntl.h>. If a shared or exclusive lock cannot be set, fcntl() will return immediately with a return value of −1.

F_SETLK64 Equivalent to F_SETLK, but takes a struct flock64 argument rather than a struct flock argument.

F_SETLKW This command is the same as F_SETLK except that if a shared or exclusive lock is blocked by other locks, the process will wait until the request can be satisfied. If a signal that is to be caught is received while fcntl() is waiting for a region, fcntl() will be interrupted. Upon return from the process' signal handler, fcntl() will return −1 with errno set to EINTR, and the lock operation will not be done.

F_SETLKW64 Equivalent to F_SETLKW, but takes a struct flock64 argument rather than a struct flock argument.

When a shared lock is set on a segment of a file, other processes will be able to set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock will fail if the file descriptor was not opened with read access.

An exclusive lock will prevent any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock will fail if the file descriptor was not opened with write access.

The flock structure contains at least the following elements:

```
short   l_type;       /* lock operation type */
short   l_whence;     /* lock base indicator */
off_t   l_start;      /* starting offset from base */
off_t   l_len;        /* lock length; l_len == 0 means
                         until end of file */
int     l_sysid;      /* system ID running process holding lock */
pid_t   l_pid;        /* process ID of process holding lock */
```

The value of l_whence is SEEK_SET, SEEK_CUR, or SEEK_END, to indicate that the relative offset l_start bytes will be measured from the start of the file, current position or end of the file, respectively. The value of l_len is the number of consecutive bytes to be locked. The value of

l_len may be negative (where the definition of off_t permits negative values of l_len). After a successful F_GETLK or F_GETLK64 request, that is, one in which a lock was found, the value of l_whence will be SEEK_SET.

The l_pid and l_sysid fields are used only with F_GETLK or F_GETLK64 to return the process ID of the process holding a blocking lock and to indicate which system is running that process.

If l_len is positive, the area affected starts at l_start and ends at l_start + l_len − 1. If l_len is negative, the area affected starts at l_start + l_len and ends at l_start − 1. Locks may start and extend beyond the current end of a file, but must not be negative relative to the beginning of the file. A lock will be set to extend to the largest possible value of the file offset for that file by setting l_len to 0. If such a lock also has l_start set to 0 and l_whence is set to SEEK_SET, the whole file will be locked.

If a process has an existing lock in which l_len is 0 and which includes the last byte of the requested segment, and an unlock (F_UNLCK) request is made in which l_len is non-zero and the offset of the last byte of the requested segment is the maximum value for an object of type off_t, then the F_UNLCK request will be treated as a request to unlock from the start of the requested segment with an l_len equal to 0. Otherwise, the request will attempt to unlock only the requested segment.

There will be at most one type of lock set for each byte in the file. Before a successful return from an F_SETLK, F_SETLK64, F_SETLKW, or F_SETLKW64 request when the calling process has previously existing locks on bytes in the region specified by the request, the previous lock type for each byte in the specified region will be replaced by the new lock type. As specified above under the descriptions of shared locks and exclusive locks, an F_SETLK, F_SETLK64, F_SETLKW, or F_SETLKW64 request will (respectively) fail or block when another process has existing locks on bytes in the specified region and the type of any of those locks conflicts with the type specified in the request.

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 created using fork(2).

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock another process' locked region. If the system detects that sleeping until a locked region is unlocked would cause a deadlock, fcntl() will fail with an EDEADLK error.

The following values for *cmd* are used for file share reservations. A share reservation is placed on an entire file to allow cooperating processes to control access to the file.

F_SHARE — Sets a share reservation on a file with the specified access mode and designates which types of access to deny.

F_UNSHARE — Remove an existing share reservation.

File share reservations are an advisory form of access control among cooperating processes, on both local and remote machines. They are most often used by DOS or Windows emulators and DOS based NFS clients. However, native UNIX versions of DOS or Windows applications may also choose to use this form of access control.

A share reservation is described by an `fshare` structure defined in <sys/fcntl.h>, which is included in <fcntl.h> as follows:

```
typedef struct fshare {
        short   f_access;
        short   f_deny;
        int     f_id;
} fshare_t;
```

A share reservation specifies the type of access, `f_access`, to be requested on the open file descriptor. If access is granted, it further specifies what type of access to deny other processes, `f_deny`. A single process on the same file may hold multiple non-conflicting reservations by specifying an identifier, `f_id`, unique to the process, with each request.

An `F_UNSHARE` request releases the reservation with the specified `f_id`. The `f_access` and `f_deny` fields are ignored.

Valid `f_access` values are:

`F_RDACC` Set a file share reservation for read-only access.

`F_WRACC` Set a file share reservation for write-only access.

`F_RWACC` Set a file share reservation for read and write access.

Valid `f_deny` values are:

`F_COMPAT` Set a file share reservation to compatibility mode.

`F_RDDNY` Set a file share reservation to deny read access to other processes.

`F_WRDNY` Set a file share reservation to deny write access to other processes.

`F_RWDNY` Set a file share reservation to deny read and write access to other processes.

`F_NODNY` Do not deny read or write access to any other process.

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

`F_DUPFD` A new file descriptor.

`F_FREESP` Value of `0`.

`F_GETFD` Value of flags defined in <fcntl.h>. The return value will not be negative.

`F_GETFL` Value of file status flags and access modes. The return value will not be negative.

F_GETLK	Value other than −1.
F_GETLK64	Value other than −1.
F_GETOWN	Value of the socket owner process or process group; this will not be −1.
F_GETXFL	Value of file status flags, access modes, and creation and assignment flags. The return value will not be negative.
F_SETFD	Value other than −1.
F_SETFL	Value other than −1.
F_SETLK	Value other than −1.
F_SETLK64	Value other than −1.
F_SETLKW	Value other than −1.
F_SETLKW64	Value other than −1.
F_SETOWN	Value other than −1.
F_SHARE	Value other than −1.
F_UNSHARE	Value other than −1.

Otherwise, −1 is returned and errno is set to indicate the error.

Errors The fcntl() function will fail if:

EAGAIN	The *cmd* argument is F_SETLK or F_SETLK64, the type of lock (l_type) is a shared (F_RDLCK) or exclusive (F_WRLCK) lock, and the segment of a file to be locked is already exclusive-locked by another process; or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process. The *cmd* argument is F_FREESP, the file exists, mandatory file/record locking is set, and there are outstanding record locks on the file; or the *cmd* argument is F_SETLK, F_SETLK64, F_SETLKW, or F_SETLKW64, mandatory file/record locking is set, and the file is currently being mapped to virtual memory using mmap(2). The *cmd* argument is F_SHARE and f_access conflicts with an existing f_deny share reservation.
EBADF	The *fildes* argument is not a valid open file descriptor; or the *cmd* argument is F_SETLK, F_SETLK64, F_SETLKW, or F_SETLKW64, the type of lock, l_type, is a shared lock (F_RDLCK), and *fildes* is not a valid file descriptor open for reading; or the type of lock l_type is an exclusive lock (F_WRLCK) and *fildes* is not a valid file descriptor open for writing.

	The *cmd* argument is F_FREESP and *fildes* is not a valid file descriptor open for writing.
	The *cmd* argument is F_DUP2FD, and *arg* is negative or is not less than the current resource limit for RLIMIT_NOFILE.
	The *cmd* argument is F_SHARE, the f_access share reservation is for write access, and *fildes* is not a valid file descriptor open for writing.
	The *cmd* argument is F_SHARE, the f_access share reservation is for read access, and *fildes* is not a valid file descriptor open for reading.
EFAULT	The *cmd* argument is F_GETLK, F_GETLK64, F_SETLK, F_SETLK64, F_SETLKW, F_SETLKW64, or F_FREESP and the *arg* argument points to an illegal address.
	The *cmd* argument is F_SHARE or F_UNSHARE and *arg* points to an illegal address.
EINTR	The *cmd* argument is F_SETLKW or F_SETLKW64 and the function was interrupted by a signal.
EINVAL	The *cmd* argument is invalid or not supported by the file system; or the *cmd* argument is F_DUPFD and *arg* is negative or greater than or equal to OPEN_MAX; or the *cmd* argument is F_GETLK, F_GETLK64, F_SETLK, F_SETLK64, F_SETLKW, or F_SETLKW64 and the data pointed to by *arg* is not valid; or *fildes* refers to a file that does not support locking.
	The *cmd* argument is F_UNSHARE and a reservation with this f_id for this process does not exist.
EIO	An I/O error occurred while reading from or writing to the file system.
EMFILE	The *cmd* argument is F_DUPFD and either OPEN_MAX file descriptors are currently open in the calling process, or no file descriptors greater than or equal to *arg* are available.
ENOLCK	The *cmd* argument is F_SETLK, F_SETLK64, F_SETLKW, or F_SETLKW64 and satisfying the lock or unlock request would result in the number of locked regions in the system exceeding a system-imposed limit.
ENOLINK	Either the *fildes* argument is on a remote machine and the link to that machine is no longer active; or the *cmd* argument is F_FREESP, the file is on a remote machine, and the link to that machine is no longer active.
EOVERFLOW	One of the values to be returned cannot be represented correctly.
	The *cmd* argument is F_GETLK, F_SETLK, or F_SETLKW and the smallest or, if l_len is non-zero, the largest, offset of any byte in the requested segment cannot be represented correctly in an object of type off_t.

The *cmd* argument is F_GETLK64, F_SETLK64, or F_SETLKW64 and the smallest or, if l_len is non-zero, the largest, offset of any byte in the requested segment cannot be represented correctly in an object of type off64_t.

The fcntl() function may fail if:

EAGAIN The *cmd* argument is F_SETLK, F_SETLK64, F_SETLKW, or F_SETLKW64, and the file is currently being mapped to virtual memory using mmap(2).

EDEADLK The *cmd* argument is F_SETLKW or F_SETLKW64, the lock is blocked by some lock from another process and putting the calling process to sleep, waiting for that lock to become free would cause a deadlock.

The *cmd* argument is F_FREESP, mandatory record locking is enabled, O_NDELAY and O_NONBLOCK are clear and a deadlock condition was detected.

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal Safe

See Also

lockd(1M), chmod(2), close(2), creat(2), dup(2), exec(2), fork(2), mmap(2), open(2), pipe(2), read(2), sigaction(2), write(2), dup2(3C), fcntl.h(3HEAD), attributes(5), standards(5)

Programming Interfaces Guide

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. The network share locking protocol does not support the f_deny value of F_COMPAT. For network file systems, if f_access is F_RDACC, f_deny is mapped to F_RDDNY. Otherwise, it is mapped to F_RWDNY.

To prevent possible file corruption, the system may reject mmap() requests for advisory locked files, or it may reject advisory locking requests for mapped files. Applications that require a file be both locked and mapped should lock the entire file (l_start and l_len both set to 0). If a file is mapped, the system may reject an unlock request, resulting in a lock that does not cover the entire file.

The process ID returned for locked files on network file systems might not be meaningful.

If the file server crashes and has to be rebooted, the lock manager (see `lockd(1M)`) attempts to recover all locks that were associated with that server. If a lock cannot be reclaimed, the process that held the lock is issued a `SIGLOST` signal.

Name fork, fork1, forkall, forkx, forkallx – create a new process

Synopsis

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

pid_t fork(void);

pid_t fork1(void);

pid_t forkall(void);

#include <sys/fork.h>

pid_t forkx(int flags);

pid_t forkallx(int flags);
```

Description The `fork()`, `fork1()`, `forkall()`, `forkx()`, and `forkallx()` functions create a new process. The address space of the new process (child process) is an exact copy of the address space 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

- task ID and project ID
- processor bindings (see processor_bind(2))
- processor set bindings (see pset_bind(2))
- process privilege sets (see getppriv(2))
- process flags (see getpflags(2))
- active contract templates (see contract(4))

Scheduling priority and any per-process scheduling parameters that are specific to a given scheduling class might or might not be inherited according to the policy of that particular class (see priocntl(2)). The child process might or might not be in the same process contract as the parent (see process(4)). 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(3C) are not inherited by the child process.
- No asynchronous input or asynchronous output operations are inherited by the child.
- Any preferred hardware address tranlsation sizes (see memcntl(2)) are inherited by the child.
- The child process holds no contracts (see contract(4)).

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

Although any open door descriptors in the parent are shared by the child, only the parent will receive a door invocation from clients even if the door descriptor is open in the child. If a descriptor is closed in the parent, attempts to operate on the door descriptor will fail even if it is still open in the child.

Threads A call to `forkall()` or `forkallx()` replicates in the child process all of the threads (see `thr_create(3C)` and `pthread_create(3C)`) in the parent process. A call to `fork1()` or `forkx()` replicates only the calling thread in the child process.

A call to `fork()` is identical to a call to `fork1()`; only the calling thread is replicated in the child process. This is the POSIX-specified behavior for `fork()`.

In releases of Solaris prior to Solaris 10, the behavior of `fork()` depended on whether or not the application was linked with the POSIX threads library. When linked with `-lthread` (Solaris Threads) but not linked with `-lpthread` (POSIX Threads), `fork()` was the same as `forkall()`. When linked with `-lpthread`, whether or not also linked with `-lthread`, `fork()` was the same as `fork1()`.

Prior to Solaris 10, either `-lthread` or `-lpthread` was required for multithreaded applications. This is no longer the case. The standard C library provides all threading support for both sets of application programming interfaces. Applications that require replicate-all fork semantics must call `forkall()` or `forkallx()`.

Fork Extensions The `forkx()` and `forkallx()` functions accept a *flags* argument consisting of a bitwise inclusive-OR of zero or more of the following flags, which are defined in the header `<sys/fork.h>`:

`FORK_NOSIGCHLD`
: Do not post a `SIGCHLD` signal to the parent process when the child process terminates, regardless of the disposition of the `SIGCHLD` signal in the parent. `SIGCHLD` signals are still possible for job control stop and continue actions if the parent has requested them.

`FORK_WAITPID`
: Do not allow wait-for-multiple-pids by the parent, as in `wait()`, `waitid(P_ALL)`, or `waitid(P_PGID)`, to reap the child and do not allow the child to be reaped automatically due the disposition of the SIGCHLD signal being set to be ignored in the parent. Only a specific wait for the child, as in `waitid(P_PID, pid)`, is allowed and it is required, else when the child exits it will remain a zombie until the parent exits.

If the *flags* argument is 0 `forkx()` is identical to `fork()` and `forkallx()` is identical to `forkall()`.

`fork()` Safety If a multithreaded application calls `fork()`, `fork1()`, or `forkx()`, and the child does more than simply call one of the `exec(2)` functions, there is a possibility of deadlock occurring in the child. The application should use `pthread_atfork(3C)` to ensure safety with respect to this deadlock. Should there be any outstanding mutexes throughout the process, the application should call `pthread_atfork()` to wait for and acquire those mutexes prior to calling `fork()`, `fork1()`, or `forkx()`. See "MT-Level of Libraries" on the `attributes(5)` manual page.

The `pthread_atfork()` mechanism is used to protect the locks that `libc(3LIB)` uses to implement interfaces such as `malloc(3C)`. All interfaces provided by `libc` are safe to use in a child process following a `fork()`, except when `fork()` is executed within a signal handler.

The POSIX standard (see `standards(5)`) requires fork to be Async-Signal-Safe (see `attributes(5)`). This cannot be made to happen with fork handlers in place, because they acquire locks. To be in nominal compliance, no fork handlers are called when `fork()` is executed within a signal context. This leaves the child process in a questionable state with respect to its locks, but at least the calling thread will not deadlock itself attempting to acquire a lock that it already owns. In this situation, the application should strictly adhere to the advice given in the POSIX specification: "To avoid errors, the child process may only execute Async-Signal-Safe operations until such time as one of the `exec(2)` functions is called."

Return Values Upon successful completion, `fork()`, `fork1()`, `forkall()`, `forkx()`, and `forkallx()` return `0` to the child process and return the process ID of the child process to the parent process. Otherwise, `(pid_t)−1` is returned to the parent process, no child process is created, and `errno` is set to indicate the error.

Errors The `fork()`, `fork1()`, `forkall()`, `forkx()`, and `forkallx()` functions will fail if:

`EAGAIN` A resource control or limit on the total number of processes, tasks or LWPs under execution by a single user, task, project, or zone has been exceeded, or the total amount of system memory available is temporarily insufficient to duplicate this process.

`ENOMEM` There is not enough swap space.

`EPERM` The {`PRIV_PROC_FORK`} privilege is not asserted in the effective set of the calling process.

The `forkx()` and `forkallx()` functions will fail if:

`EINVAL` The *flags* argument is invalid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe.
Standard	See below.

For `fork()`, see `standards(5)`.

See Also alarm(2), exec(2), exit(2), fcntl(2), getitimer(2), getrlimit(2), memcntl(2), mmap(2), nice(2), priocntl(2), semop(2), shmop(2), times(2), umask(2), waitid(2), door_create(3C), exit(3C), plock(3C), pthread_atfork(3C), pthread_create(3C), signal(3C), system(3C), thr_create(3C) timer_create(3C), wait(3C), contract(4), process(4), attributes(5), privileges(5), standards(5)

Notes An application should call `_exit()` rather than exit(3C) if it cannot `execve()`, since `exit()` will flush and close standard I/O channels and thereby corrupt the parent process's standard I/O data structures. Using exit(3C) will flush buffered data twice. See exit(2).

The thread in the child that calls `fork()`, `fork1()`, or `forkx()` must not depend on any resources held by threads that no longer exist in the child. In particular, locks held by these threads will not be released.

In a multithreaded process, `forkall()` in one thread can cause blocking system calls to be interrupted and return with an `EINTR` error.

Name fpathconf, pathconf – get configurable pathname variables

Synopsis

```
#include <unistd.h>

long fpathconf(int fildes, int name);

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

Description The `fpathconf()` and `pathconf()` functions determine the current value of a configurable limit or option (variable) that is associated with a file or directory.

For `pathconf()`, the *path* argument points to the pathname of a file or directory.

For `fpathconf()`, the *fildes* argument is an open file descriptor.

The *name* argument represents the variable to be queried relative to that file or directory. The variables in the following table come from <`limits.h`> or <`unistd.h`> and the symbolic constants, defined in <`unistd.h`>, are the corresponding values used for *name:*

Variable	Value of *name*	Notes
{ACL_ENABLED}	_PC_ACL_ENABLED	10
{FILESIZEBITS}	_PC_FILESIZEBITS	3,4
{LINK_MAX}	_PC_LINK_MAX	1
{MAX_CANON}	_PC_MAX_CANON	2
{MAX_INPUT}	_PC_MAX_INPUT	2
{MIN_HOLE_SIZE}	_PC_MIN_HOLE_SIZE	11
{NAME_MAX}	_PC_NAME_MAX	3, 4
{PATH_MAX}	_PC_PATH_MAX	4,5
{PIPE_BUF}	_PC_PIPE_BUF	6
{POSIX_ALLOC_SIZE_MIN}	_PC_ALLOC_SIZE_MIN	
{POSIX_REC_INCR_XFER_SIZE}	_PC_REC_INCR_XFER_SIZE	
{POSIX_REC_MAX_XFER_SIZE}	_PC_REC_MAX_XFER_SIZE	
{POSIX_REC_MIN_XFER_SIZE}	_PC_REC_MIN_XFER_SIZE	
{POSIX_REC_XFER_ALIGN}	_PC_REC_XFER_ALIGN	
{SYMLINK_MAX}	_PC_SYMLINK_MAX	4, 9
{XATTR_ENABLED}	_PC_XATTR_ENABLED	1

Variable	Value of *name*	Notes
{SATTR_ENABLED}	_PC_SATTR_ENABLED	
{XATTR_EXISTS}	_PC_XATTR_EXISTS	1
{SATTR_EXISTS}	_PC_SATTR_EXISTS	
{ACCESS_FILTERING}	_PC_ACCESS_FILTERING	12
_POSIX_CHOWN_RESTRICTED	_PC_CHOWN_RESTRICTED	7
_POSIX_NO_TRUNC	_PC_NO_TRUNC	3, 4
_POSIX_VDISABLE	_PC_VDISABLE	2
_POSIX_ASYNC_IO	_PC_ASYNC_IO	8
_POSIX_PRIO_IO	_PC_PRIO_IO	8
_POSIX_SYNC_IO	_PC_SYNC_IO	8
_POSIX_TIMESTAMP_RESOLUTION	_PC_TIMESTAMP_RESOLUTION	1

Notes:

1. If *path* or *fildes* refers to a directory, the value returned applies to the directory itself.
2. If *path* or *fildes* does not refer to a terminal file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
3. If *path* or *fildes* refers to a directory, the value returned applies to filenames within the directory.
4. If *path* or *fildes* does not refer to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.
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* refers to a FIFO, or *fildes* refers to a pipe or FIFO, the value returned applies to the referenced object. If *path* or *fildes* refers to a directory, the value returned applies to any FIFO that exists or can be created within the directory. If *path* or *fildes* refers to any other type of file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
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.
8. If *path* or *fildes* refers to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.
9. If *path* or *fildes* refers to a directory, the value returned is the maximum length of the string that a symbolic link in that directory can contain.

10. If *path* or *fildes* refers to a file or directory in a file system that supports ACLs, the value returned is the bitwise inclusive OR of the following flags associated with ACL types supported by the file system; otherwise 0 is returned.

    `_ACL_ACE_ENABLED` The file system supports ACE ACLs.

    `_ACL_ACLENT_ENABLED` The file system supports UFS aclent ACLs.

11. If a filesystem supports the reporting of holes (see `lseek(2)`, `pathconf()` and `fpathconf()` return a positive number that represents the minimum hole size returned in bytes. The offsets of holes returned will be aligned to this same value. A special value of 1 is returned if the filesystem does not specify the minimum hole size but still reports holes.
12. If *path* or *fildes* refers to a directory and the file system in which the directory resides supports access filtering, a non-zero value is returned. Otherwise, 0 is returned.

Return Values If *name* is an invalid value, both `pathconf()` and `fpathconf()` return `−1` and `errno` is set to indicate the error.

If the variable corresponding to *name* has no limit for the *path* or file descriptor, both `pathconf()` and `fpathconf()` return `−1` without changing `errno`. If `pathconf()` needs to use *path* to determine the value of *name* and `pathconf()` does not support the association of *name* with the file specified by *path*, or if the process did not have appropriate privileges to query the file specified by *path*, or *path* does not exist, `pathconf()` returns `−1` and `errno` is set to indicate the error.

If `fpathconf()` needs to use *fildes* to determine the value of *name* and `fpathconf()` does not support the association of *name* with the file specified by *fildes*, or if *fildes* is an invalid file descriptor, `fpathconf()` returns `−1` and `errno` is set to indicate the error.

Otherwise `pathconf()` or `fpathconf()` returns the current variable value for the file or directory without changing `errno`. The value returned will not be more restrictive than the corresponding value available to the application when it was compiled with `<limits.h>` or `<unistd.h>`.

Errors The `pathconf()` function will fail if:

`EINVAL` The value of *name* is not valid.

`ELOOP` A loop exists in symbolic links encountered during resolution of the *path* argument.

The `fpathconf()` function will fail if:

`EINVAL` The value of *name* is not valid.

The `pathconf()` function may fail if:

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

EINVAL	An association of the variable *name* with the specified file is not supported.
ENAMETOOLONG	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
ENAMETOOLONG	As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.
ENOENT	A component of *path* does not name an existing file or *path* is an empty string.
ENOTDIR	A component of the path prefix is not a directory.

The `fpathconf()` function may fail if:

EBADF	The *fildes* argument is not a valid file descriptor.
EINVAL	An association of the variable *name* with the specified file is not supported.

Usage The {SYMLINK_MAX} variable applies only to the `fpathconf()` function.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See `standards(5)`.

See Also `lseek(2)`, `confstr(3C)`, `limits.h(3HEAD)`, `sysconf(3C)`, `attributes(5)`, `standards(5)`

Name futimens, utimensat – set file access and modification times

Synopsis

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

int futimens(int fd, const struct timespec times[2]);

int utimensat(int fd, const char *path,
     const struct timespec times[2], int flag);
```

Description The `futimens()` and `utimensat()` functions set the access and modification times of a file to the values of the *times* argument. The `futimens()` function changes the times of the file associated with the file descriptor *fd*. The `utimensat()` function changes the times of the file pointed to by the path argument, relative to the directory associated with the file descriptor *fd*. Both functions allow time specifications accurate to the nanosecond.

The *times* argument is an array of two `timespec` structures. The first array member represents the date and time of last access, and the second member represents the date and time of last modification. The times in the `timespec` structure are measured in seconds and nanoseconds since the Epoch. The file's relevant timestamp is set to the greatest value supported by the file system that is not greater than the specified time.

If the *tv_nsec* field of a `timespec` structure has the special value `UTIME_NOW`, the file's relevant timestamp is set to the greatest value supported by the file system that is not greater than the current time. If the *tv_nsec* field has the special value `UTIME_OMIT`, the file's relevant timestamp is not changed. In either case, the *tv_sec* field is ignored.

If the times argument is a null pointer, both the access and modification timestamps are set to the greatest value supported by the file system that is not greater than the current time. If `utimensat()` is passed a relative path in the path argument, the file to be used is relative to the directory associated with the file descriptor *fd* instead of the current working directory.

If `utimensat()` is passed the special value `AT_FDCWD` in the *fd* parameter, the current working directory is used.

Only a process with the effective user ID equal to the user ID of the file, or with write access to the file, or with appropriate privileges may use `futimens()` or `utimensat()` with a null pointer as the times argument or with both *tv_nsec* fields set to the special value `UTIME_NOW`. Only a process with the effective user ID equal to the user ID of the file or with appropriate privileges may use `futimens()` or `utimensat()` with a non-null times argument that does not have both *tv_nsec* fields set to `UTIME_NOW` and does not have both *tv_nsec* fields set to `UTIME_OMIT`. If both *tv_nsec* fields are set to `UTIME_OMIT`, no ownership or permissions check is performed for the file, but other error conditions are still detected (including `EACCES` errors related to the path prefix).

Values for the flag argument of `utimensat()` are constructed by a bitwise-inclusive OR of flags from the following list, defined in `<fcntl.h>`:

AT_SYMLINK_NOFOLLOW
If path names a symbolic link, then the access and modification times of the symbolic link are changed.

Upon completion, `futimens()` and `utimensat()` mark the last file status change timestamp for update.

Return Values Upon successful completion, these functions return 0. Otherwise, these functions return -1 and set `errno` to indicate the error. If -1 is returned, the file times are not affected.

Errors The `futimens()` and `utimensat()` functions will fail if:

`EACCES` The times argument is a null pointer, or both *tv_nsec* values are `UTIME_NOW`, and the effective user ID of the process does not match the owner of the file and write access is denied.

`EINVAL` Either of the times argument structures specified a *tv_nsec* value that was neither `UTIME_NOW` nor `UTIME_OMIT`, and was a value less than zero or greater than or equal to 1000 million.

A new file timestamp would be a value whose *tv_sec* component is not a value supported by the file system.

`EPERM` The times argument is not a null pointer, does not have both *tv_nsec* fields set to `UTIME_NOW`, does not have both *tv_nsec* fields set to `UTIME_OMIT`, the calling process' effective user ID has write access to the file but does not match the owner of the file, and the calling process does not have appropriate privileges.

`EROFS` The file system containing the file is read-only.

The `futimens()` function will fail if:

`EBADF` The *fd* argument is not a valid file descriptor.

The `utimensat()` function will fail if:

`EACCES` The permissions of the directory underlying *fd* do not permit directory searches.

`EBADF` The path argument does not specify an absolute path and the *fd* argument is neither `AT_FDCWD` nor a valid file descriptor open for reading.

`ENOTDIR` The path argument is not an absolute path and *fd* is neither `AT_FDCWD` nor a file descriptor associated with a directory.

`EACCES` Search permission is denied by a component of the path prefix.

`ELOOP` Too many symbolic links were encountered during resolution of the path argument.

ENAMETOOLONG The length of the path argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

ENOENT A component of path does not name an existing file or path is an empty string.

ENOTDIR A component of the path prefix is not a directory, or the path argument contains at least one character that is not a slash (/) and ends with one or more trailing slash characters and the last pathname component names an existing file that is neither a directory nor a symbolic link to a directory.

The `utimensat()` function will fail if:

ENAMETOOLONG Path name resolution of a symbolic link produced an intermediate result with a length that exceeds {PATH_MAX}.

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe

See Also

stat(2), utime(2), utimes(2), attributes(5), fsattr(5)

Name getacct, putacct, wracct – get, put, or write extended accounting data

Synopsis

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

size_t getacct(idtype_t idtype, id_t id, void *buf, size_t bufsize);

int putacct(idtype_t idtype, id_t id, void *buf, size_t bufsize, int flags);

int wracct(idtype_t idtype, id_t id, int flags);
```

Description These functions provide access to the extended accounting facility.

The `getacct()` function returns extended accounting buffers from the kernel for currently executing tasks and processes. The resulting data buffer is a packed `exacct` object that can be unpacked using `ea_unpack_object()` (see `ea_pack_object(3EXACCT)`) and subsequently manipulated using the functions of the extended accounting library, `libexacct(3LIB)`.

The `putacct()` function provides privileged processes the ability to tag accounting records with additional data specific to that process. For instance, a queueing facility might want to record to which queue a given task or process was submitted prior to running. The *flags* argument determines whether the contents of *buf* should be treated as raw data (`EP_RAW`) or as an embedded `exacct` structure (`EP_EXACCT_OBJECT`). In the case of `EP_EXACCT_OBJECT`, *buf* must be a packed `exacct` object as returned by `ea_pack_object(3EXACCT)`. The use of an inappropriate flag or the inclusion of corrupt `exacct` data will likely corrupt the enclosing `exacct` file.

The `wracct()` function requests the kernel to write, given its internal state of resource usage, the appropriate data for the specified task or process. The flags field determines whether a partial (`EW_PARTIAL`) or interval record (`EW_INTERVAL`) is written.

These functions require root privilege, as they allow inquiry or reporting relevant to system tasks and processes other than the invoking process. The `putacct()` and `wracct()` functions also cause the kernel to write records to the system's extended accounting files.

Return Values The `getacct()` function returns the number of bytes required to represent the extended accounting record for the requested system task or process. If *bufsize* exceeds the returned size, *buf* will contain a valid accounting record buffer. If *bufsize* is less than the return value, *buf* will contain the first *bufsize* bytes of the record. If *bufsize* is 0, `getacct()` returns only the number of bytes required to represent the extended accounting record. In the event of failure, `−1` is returned and `errno` is set to indicate the error.

The `putacct()` and `wracct()` functions return `0` if the record was successfully written. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `getacct()`, `putacct()`, and `wracct()` functions will fail if:

`EINVAL` The *idtype* argument was not `P_TASKID` or `P_PID`.

ENOSPC The file system containing the extended accounting file is full. The `wracct()` or `putacct()` function will fail if the record size would exceed the amount of space remaining on the file system.

ENOTACTIVE The extended accounting facility for the requested `idtype_t` is not active. Either `putacct()` attempted to write a task record when the task accounting file was unset, or `getacct()` attempted to retrieve accounting data for a process when extended process accounting was inactive.

EPERM The {`PRIV_SYS_ACCT`} privilege is not asserted in the effective set of the calling process.

ERSCH The *id* argument does not refer to a presently active system task ID or process ID.

The `putacct()` and `wracct()` functions will fail if:

EINVAL The *flags* argument is neither `EW_PARTIAL` nor `EW_INTERVAL`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

See Also `ea_pack_object(3EXACCT)`, `libexacct(3LIB)attributes(5)`

Name getaudit, setaudit, getaudit_addr, setaudit_addr – get or set process audit information

Synopsis

```
cc [ flag ... ] file ... -lbsm  -lsocket   -lnsl   [ library ... ]
#include <sys/param.h>
#include <bsm/libbsm.h>

int getaudit(struct auditinfo *info);

int setaudit(struct auditinfo *info);

int getaudit_addr(struct auditinfo_addr *info, int length);

int setaudit_addr(struct auditinfo_addr *info, int length);
```

Description The `getaudit()` function gets the audit ID, the preselection mask, the terminal ID and the audit session ID for the current process.

The `getaudit()` function can fail and return an `E2BIG` errno if the address field in the terminal ID is larger than 32 bits. In this case, `getaudit_addr()` should be used.

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

The `getaudit_addr()` function returns a variable length `auditinfo_addr` structure that contains the audit ID, the preselection mask, the terminal ID, and the audit session ID for the current process. The terminal ID contains a size field that indicates the size of the network address.

The `setaudit_addr()` function sets the audit ID, the preselection mask, the terminal ID, and the audit session ID for the current process. The values are taken from the variable length struture `auditinfo_addr`. The terminal ID contains a size field that indicates the size of the network address.

The `auditinfo` structure is used to pass the process audit information and 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 */
```

The `auditinfo_addr` structure is used to pass the process audit information and contains the following members:

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

Return Values Upon successful completion, `getaudit()` and `setaudit()` return `0`. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `getaudit()` and `setaudit()` functions will fail if:

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

`EPERM` The {`PRIV_SYS_AUDIT`} privilege is not asserted in the effective set of the calling process.

Usage The calling process must have the {`PRIV_SYS_AUDIT`} privilege asserted in its effective set.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Stable
MT-Level	MT-Safe

See Also `bsmconv(1M)`, `audit(2)`, `attributes(5)`

Notes The functionality described in this man page is available only if the Solaris Auditing has been enabled. See `bsmconv(1M)` for more information.

Name getauid, setauid – get or set user audit identity

Synopsis

```
cc [ flag ... ] file ... -lbsm  -lsocket   -lnsl   [ library ... ]
#include <sys/param.h>
#include <bsm/libbsm.h>

int getauid(au_id_t *auid);

int setauid(au_id_t *auid);
```

Description The `getauid()` function 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()` function sets the audit user ID for the current process.

Return Values Upon successful completion, the `getauid()` function returns the audit user ID of the current process on success. Otherwise, it returns −1 and sets `errno` to indicate the error.

Upon successful completion the `setauid()` function returns `0`. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors The `getauid()` and `setauid()` functions will fail if:

`EFAULT` The *auid* argument points to an invalid address.

`EPERM` The {`PRIV_SYS_AUDIT`} privilege is not asserted in the effective set of the calling process.

The `getauid()` function will fail if:

`EPERM` The {`PRIV_PROC_AUDIT`} privilege is not asserted in the effective set of the calling process.

Usage Only a process with appropriate privileges can successfully execute these calls.

See Also `bsmconv(1M)`, `audit(2)`, `getaudit(2)`, `privileges(5)`

Notes The functionality described on this manual page is available only if the Solaris Auditing has been enabled. See `bsmconv(1M)` for more information.

These functions have been superseded by `getaudit(2)` and `setaudit()`.

Name getcontext, setcontext – get and set current user context

Synopsis

```
#include <ucontext.h>

int getcontext(ucontext_t *ucp);

int setcontext(const ucontext_t *ucp);
```

Description The `getcontext()` function initializes the structure pointed to by *ucp* to the current user context of the calling process. The `ucontext_t` type that *ucp* points to defines the user context and includes the contents of the calling process' machine registers, the signal mask, and the current execution stack.

The `setcontext()` function restores the user context pointed to by *ucp*. A successful call to `setcontext()` does not return; program execution resumes at the point specified by the *ucp* argument passed to `setcontext()`. The *ucp* argument should be created either by a prior call to `getcontext()`, or by being passed as an argument to a signal handler. If the *ucp* argument was created with `getcontext()`, program execution continues as if the corresponding call of `getcontext()` had just returned. If the *ucp* argument was created with `makecontext(3C)`, program execution continues with the function passed to `makecontext(3C)`. When that function returns, the process continues as if after a call to `setcontext()` with the *ucp* argument that was input to `makecontext(3C)`. If the *ucp* argument was passed to a signal handler, program execution continues with the program instruction following the instruction interrupted by the signal. If the `uc_link` member of the `ucontext_t` structure pointed to by the *ucp* argument is equal to 0, then this context is the main context, and the process will exit when this context returns. The effects of passing a *ucp* argument obtained from any other source are unspecified.

Return Values On successful completion, `setcontext()` does not return and `getcontext()` returns `0`. Otherwise, `−1` is returned.

Errors No errors are defined.

Usage Portable applications should not modify or access the `uc_mcontext` member of `ucontext_t`. A portable application cannot assume that context includes any process-wide static data, possibly including `errno`. Users manipulating contexts should take care to handle these explicitly when required.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

See Also `sigaction(2)`, `sigaltstack(2)`, `sigprocmask(2)`, `bsd_signal(3C)`, `makecontext(3C)`, `ucontext.h(3HEAD)`, `attributes(5)`, `standards(5)`

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

Synopsis

```
#include <dirent.h>

int getdents(int fildes, struct dirent *buf, size_t nbyte);
```

Description The `getdents()` function 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 lengths, in most cases the actual number of bytes returned will be less than *nbyte*. The file system independent directory entry is specified by the `dirent` structure. See `dirent.h`(3HEAD).

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.

Return Values Upon successful completion, a non-negative integer is returned indicating the number of bytes actually read. A return value of `0` indicates the end of the directory has been reached. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `getdents()` function will fail if:

`EBADF`	The *fildes* argument is not a valid file descriptor open for reading.
`EFAULT`	The *buf* argument points to an illegal address.
`EINVAL`	The *nbyte* argument 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`	The *fildes* argument points to a remote machine and the link to that machine is no longer active.
`ENOTDIR`	The *fildes* argument is not a directory.
`EOVERFLOW`	The value of the `dirent` structure member `d_ino` or `d_off` cannot be represented in an `ino_t` or `off_t`.

Usage The `getdents()` function was developed to implement the `readdir`(3C) function and should not be used for other purposes.

The `getdents()` function has a transitional interface for 64-bit file offsets. See `lf64`(5).

See Also `readdir`(3C), `dirent.h`(3HEAD), `lf64`(5)

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);
```

Description The `getgroups()` function 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 larger 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*.

The `setgroups()` function 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`.

Return Values Upon successful completion, `getgroups()` returns the number of supplementary group IDs set for the calling process and `setgroups()` returns `0`. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `getgroups()` and `setgroups()` functions will fail if:

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

The `getgroups()` function 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.

The `setgroups()` function will fail if:

`EINVAL` The value of *ngroups* is greater than {`NGROUPS_MAX`}.

`EPERM` The {`PRIV_PROC_SETID`} privilege is not asserted in the effective set of the calling process.

Usage Use of the `setgroups()` function requires the {`PRIV_PROC_SETID`} privilege.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	For `getgroups()`, see `standards(5)`.

See Also groups(1), chown(2), getuid(2), setuid(2), getgrnam(3C), initgroups(3C), attributes(5), privileges(5), standards(5)

Name getisax – extract valid instruction set extensions

Synopsis

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

uint_t getisax(uint32_t *array, uint_t n);
```

Description The `getisax()` function sets the vector *array* of *n* 32–bit integers to contain the bits from the `AV_`*xxx_yyy* namespace of the given instruction set architecture.

Values for `AV_`*xxx_yyy* for SPARC and SPARCV9, and their associated descriptions, can be found in `<sys/auxv_SPARC.h>`.

Values for `AV_`*xxx_yyy* for i386 and AMD64, and their associated descriptions, can be found in `<sys/auxv_386.h>`.

Return Values The `getisax()` function returns the number of array elements that contain non-zero values.

Examples **EXAMPLE 1** Use `getisax()` to determine if the SSE2 instruction set is present.

In the following example, if the message is written, the SSE2 instruction set is present and fully supportred by the operating system.

```
uint_t ui;

(void) getisax(&ui, 1);

if (ui & AV_386_SSE2)
        printf("SSE2 instruction set extension is present.\n");
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Safe

See Also `isainfo(1)`, `ld(1)`, `pargs(1)`, `attributes(5)`

Linker and Libraries Guide

SPARC Assembly Language Reference Manual

x86 Assembly Language Reference Manual

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);
```

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()` function 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 */
```

The `it_value` member indicates the time to the next timer expiration. The `it_interval` member specifies a value to be used in reloading `it_value` when the timer expires. Setting `it_value` to 0 disables a timer, regardless of the value of `it_interval`. Setting `it_interval` to 0 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 as follows:

`ITIMER_REAL` Decrements in real time. A `SIGALRM` signal is delivered to the process when this timer expires.

`ITIMER_VIRTUAL` Decrements in lightweight process (lwp) virtual time. It runs only when the calling lwp is executing. A `SIGVTALRM` signal is delivered to the calling lwp when it expires.

`ITIMER_PROF` Decrements both in lightweight process (lwp) virtual time and when the system is running on behalf of the lwp. 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 to the calling lwp. Because this signal may interrupt in-progress functions, programs using this timer must be prepared to restart interrupted functions.

`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>.

Unlike the other interval timers, the ITIMER_REALPROF interval timer is not inherited across a call to one of the exec(2) family of functions.

Return Values Upon successful completion, 0 is returned. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The getitimer() and setitimer() functions 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* argument is unrecognized.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also alarm(2), exec(2), gettimeofday(3C), sleep(3C), sysconf(3C), attributes(5), standards(5)

Notes The setitimer() function is independent of the alarm(2) and sleep(3C) functions.

The ITIMER_PROF and ITIMER_REALPROF timers 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 getlabel, fgetlabel – get file sensitivity label

Synopsis

```
cc [flags...] file... -ltsol [library...]

#include <tsol/label.h>

int getlabel(const char *path, m_label_t *label_p);

int fgetlabel(int fd, m_label_t *label_p);
```

Description The `getlabel()` function obtains the sensitivity label of the file that is named by *path*. Discretionary read, write or execute permission to the final component of *path* is not required, but all directories in the path prefix of *path* must be searchable.

The `fgetlabel()` function obtains the label of an open file that is referred to by the argument descriptor, such as would be obtained by an `open(2)` call.

The *label_p* argument is a pointer to an opaque label structure. The caller must allocate space for *label_p* by using `m_label_alloc(3TSOL)`.

Return Values Upon successful completion, `getlabel()` and `fgetlabel()` return 0. Otherwise they return −1 and set `errno` to indicate the error.

Errors The `getlabel()` function will fail if:

`EACCES`	Search permission is denied for a component of the path prefix of *path*. To override this restriction, the calling process can assert the `PRIV_FILE_DAC_SEARCH` privilege.
`EFAULT`	*label_p* or *path* points to an invalid address.
`EIO`	An I/O error occurred while reading from or writing to the file system.
`ELOOP`	Too many symbolic links were encountered in translating *path*.
`ENAMETOOLONG`	The length of the path argument exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect (see `pathconf(2)`).
`ENOENT`	The file referred to by *path* does not exist.
`ENOTDIR`	A component of the path prefix of *path* is not a directory.

The `fgetlabel()` function will fail if:

`EBADF`	The *fd* argument is not a valid open file descriptor.
`EFAULT`	The *label_p* argument points to an invalid address.
`EIO`	An I/O error occurred while reading from or writing to the file system.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcslr
Interface Stability	Committed

See Also open(2), pathconf(2), m_label_alloc(3TSOL), attributes(5), labels(5)

"Obtaining a File Label" in *Solaris Trusted Extensions Developer's Guide*

Notes The functionality described on this manual page is available only if the system is configured with Trusted Extensions.

Name getmsg, getpmsg – get next message off a stream

Synopsis

```
#include <stropts.h>

int getmsg(int fildes, struct strbuf *restrict ctlptr,
     struct strbuf *restrict dataptr, int *restrict flagsp);

int getpmsg(int fildes, struct strbuf *restrict ctlptr,
     struct strbuf *restrict dataptr, int *restrict bandp,
     int *restrict flagsp);
```

Description The `getmsg()` function 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 `getpmsg()` function behaved like `getmsg()`, but provides finer control over the priority of the messages received. Except where noted, all information pertaining to `getmsg()` also pertains to `getpmsg()`.

The *fildes* argument specifies a file descriptor referencing an open stream. The *ctlptr* and *dataptr* arguments 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 */
```

The `buf` member points to a buffer into which the data or control information is to be placed, and the `maxlen` member indicates the maximum number of bytes this buffer can hold. On return, the `len` member contains the number of bytes of data or control information actually received; 0 if there is a zero-length control or data part; or −1 if no data or control information is present in the message. The *flagsp* argument should point to an integer that indicates the type of message the user is able to receive, as described below.

The *ctlptr* argument holds the control part from the message and the *dataptr* argument holds the data part from the message. If *ctlptr* (or *dataptr*) is `NULL` or the `maxlen` member 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` member 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` member 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` member 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. A user may, however, choose to retrieve only high priority messages by setting the integer pointed to 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 to 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 to 0 otherwise.

For `getpmsg()`, the *flagsp* argument 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` member of *ctlptr* and *dataptr*.

Return Values Upon successful completion, a non-negative value is returned. A return 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 been received by 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 The `getmsg()` and `getpmsg()` functions will fail if:

`EAGAIN` The `O_NDELAY` or `O_NONBLOCK` flag is set and no messages are available.

`EBADF` The *fildes* argument is not a valid file descriptor open for reading.

`EBADMSG` Queued message to be read is not valid for `getmsg`.

`EFAULT` The *ctlptr*, *dataptr*, *bandp*, or *flagsp* argument points to an illegal address.

`EINTR` A signal was caught during the execution of 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*.

The `getmsg()` function 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.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also `Intro(2)`, `poll(2)`, `putmsg(2)`, `read(2)`, `write(2)`, `attributes(5)`, `standards(5)`

STREAMS Programming Guide

Name getpflags, setpflags – get or set process flags

Synopsis

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

uint_t getpflags(uint_t flag);

int setpflags(uint_t flag, uint_t value);
```

Description The `getpflags()` and `setpflags()` functions obtain and modify the current per-process flags.

The following values for *flag* are supported:

`PRIV_AWARE`
: This one bit flag takes the value of 0 (unset) or 1 (set). Only if this flag is set is the current process privilege-aware. A process can attempt to unset this flag but might fail silently if the observed set invariance condition cannot be met. Setting this flag is always successful. See `privileges(5)` for a discussion of this flag.

`PRIV_AWARE_RESET`
: This one bit flag takes the value of 0 (unset) or 1 (set). This causes a process to pretend it is non- privilege aware. The effective and permitted privilege set change on the change of the effective uid. When all the uid sets become the same through `setuid`(*uid*) or through `setreuid`(*uid*, *uid*), the effective and permitted set are set to the intersection between the limit set and the inheritable set. At that point, both `PRIV_AWARE` and `PRIV_AWARE_RESET` are unset.

  This flag gets automatically reset when a file becomes privilege aware, either through calling `setppriv(2)` or by setting `PRIV_AWARE` to 1.

`PRIV_DEBUG`
: This one bit flag takes the value of 0 (unset) or 1 (set). Only if this flag is set does the current process have privilege debugging enabled. Processes can set and unset this flag at will.

`NET_MAC_AWARE`
`NET_MAC_AWARE_INHERIT`
: These flags are available only if the system is configured with Trusted Extensions. These one bit flags each take the value of 0 (unset) or 1 (set). If the `NET_MAC_AWARE` flag is set then the current process is allowed to communicate with peers at labels that are different than its own, subject to MAC policy.

  The `NET_MAC_AWARE_INHERIT` flag controls the propagation of the `NET_MAC_AWARE` flag. When a process performs one of the `exec(2)` functions, the `NET_MAC_AWARE` flag is unset unless the `NET_MAC_AWARE_INHERIT` is set. `NET_MAC_AWARE_INHERIT` is always unset on one of the `exec` functions. The `PRIV_NET_MAC_AWARE` privilege is required to set either of these flags.

Return Values The `getpflags()` returns the value associated with a given per-process flag. If the *flag* argument is invalid, (`uint_t`)-1 is returned and `errno` is set to indicate the error.

Upon successful completion, `setpflags()` returns 0. Otherwise, -1 is returned and `errno` is set to indicate the error.

Errors The `getpflags()` and `setpflags()` functions will fail if:

`EINVAL` The value of *flag* or the value to which the *flag* is set is out of range.

The `setpflags()` function will fail if:

`EPERM` An attempt was made to unset `PRIV_AWARE` but the observed set invariance condition was not met.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe

See Also `ppriv(1)`, `setppriv(2)`, `attributes(5)`, `privileges(5)`

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

Synopsis

```
#include <unistd.h>

pid_t getpid(void);

pid_t getpgrp(void);

pid_t getppid(void);

pid_t getpgid(pid_t pid);
```

Description The `getpid()` function returns the process ID of the calling process.

The `getpgrp()` function returns the process group ID of the calling process.

The `getppid()` function returns the parent process ID of the calling process.

The `getpgid()` function 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 0.

Return Values The `getpid()`, `getpgrp()`, and `getppid()` functions are always successful and no return value is reserved to indicate an error.

Upon successful completion, `getpgid()` returns the process group ID. Otherwise, `getpgid()` returns `(pid_t)−1` and sets `errno` to indicate the error.

Errors The `getpgid()` function will fail if:

`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*.

The `getpgid()` function may fail if:

`EINVAL` The value of the *pid* argument is invalid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See `standards(5)`.

See Also Intro(2), exec(2), fork(2), getsid(2), setpgid(2), setpgrp(2), setsid(2), signal(3C), attributes(5), standards(5)

Name getppriv, setppriv – get or set a privilege set

Synopsis

```
#include <priv.h>

int getppriv(priv_ptype_t which, priv_set_t *set);

int setppriv(priv_op_t op, priv_ptype_t which, priv_set_t *set);
```

Description The `getppriv()` function returns the process privilege set specified by *which* in the set pointed to by *set*. The memory for *set* is allocated with `priv_allocset()` and freed with `priv_freeset()`. Both functions are documented on the `priv_addset(3C)` manual page.

The `setppriv()` function sets or changes the process privilege set. The *op* argument specifies the operation and can be one of `PRIV_OFF`, `PRIV_ON` or `PRIV_SET`. The *which* argument specifies the name of the privilege set. The *set* argument specifies the set.

If *op* is `PRIV_OFF`, the privileges in *set* are removed from the process privilege set specified by *which*. There are no restrictions on removing privileges from process privileges sets, but the following apply:

- Privileges removed from `PRIV_PERMITTED` are silently removed from `PRIV_EFFECTIVE`.
- If privileges are removed from `PRIV_LIMIT`, they are not removed from the other sets until one of `exec(2)` functions has successfully completed.

If *op* is `PRIV_ON`, the privileges in *set* are added to the process privilege set specified by *which*. The following operations are permitted:

- Privileges in `PRIV_PERMITTED` can be added to `PRIV_EFFECTIVE` without restriction.
- Privileges in `PRIV_PERMITTED` can be added to `PRIV_INHERITABLE` without restriction.
- All operations that attempt to add privileges that are already present are permitted.

If *op* is `PRIV_SET`, the privileges in *set* replace completely the process privilege set specified by *which*. `PRIV_SET` is implemented in terms of `PRIV_OFF` and `PRIV_ON`. The same restrictions apply.

Return Values Upon successful completion, 0 is returned. Otherwise, -1 is returned and `errno` is set to indicate the error.

Errors The `getppriv()` and `setppriv()` functions will fail if:

`EINVAL` The value of *op* or *which* is out of range.

`EFAULT` The *set* argument points to an illegal address.

The `setppriv()` function will fail if:

`EPERM` The application attempted to add privileges to `PRIV_LIMIT` or `PRIV_PERMITTED`, or the application attempted to add privileges to `PRIV_INHERITABLE` or `PRIV_EFFECTIVE` which were not in `PRIV_PERMITTED`.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also priv_addset(3C), attributes(5), privileges(5)

Name getrlimit, setrlimit – control maximum system resource consumption

Synopsis

```
#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 the `getrlimit()` and set with `setrlimit()` functions.

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 {`PRIV_SYS_RESOURCE`} asserted in the effective set 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`. The *rlp* argument is a pointer to `struct rlimit` that includes the following members:

```
rlim_t    rlim_cur;     /* current (soft) limit */
rlim_t    rlim_max;     /* hard limit */
```

The type `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 as follows:

`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. The `SIGXCPU` signal 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. The `brk(2)` function 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. The `SIGXFSZ` signal 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.

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. To guarantee that the process stack can grow to the limit, the limit must be altered prior to the execution of the process in which the new stack size is to be used.

Within a multithreaded process, `setrlimit()` has no impact on the stack size limit for the calling thread if the calling thread is not the main thread. A call to `setrlimit()` for `RLIMIT_STACK` impacts only the main thread's stack, and should be made only from the main thread, if at all.

The `SIGSEGV` signal 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. If this limit is exceeded, the `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.

`RLIMIT_AS` This is the maximum size of a process's total available memory, in bytes. If this limit is exceeded, the `brk(2)`, `malloc(3C)`, `mmap(2)` and `sbrk(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`

When using the `getrlimit()` function, if a resource limit can be represented correctly in an object of type `rlim_t`, then its representation is returned; otherwise, if the value of the resource limit is equal to that of the corresponding saved hard limit, the value returned is `RLIM_SAVED_MAX`; otherwise the value returned is `RLIM_SAVED_CUR`.

When using the `setrlimit()` function, if the requested new limit is `RLIM_INFINITY`, the new limit will be "no limit"; otherwise if the requested new limit is `RLIM_SAVED_MAX`, the new limit will be the corresponding saved hard limit; otherwise, if the requested new limit is `RLIM_SAVED_CUR`, the new limit will be the corresponding saved soft limit; otherwise, the new limit will be the requested value. In addition, if the corresponding saved limit can be represented correctly in an object of type `rlim_t`, then it will be overwritten with the new limit.

The result of setting a limit to `RLIM_SAVED_MAX` or `RLIM_SAVED_CUR` is unspecified unless a previous call to `getrlimit()` returned that value as the soft or hard limit for the corresponding resource limit.

A limit whose value is greater than `RLIM_INFINITY` is permitted.

The `exec` family of functions also cause resource limits to be saved. See `exec(2)`.

Return Values Upon successful completion, `getrlimit()` and `setrlimit()` return `0`. Otherwise, these functions return `−1` and set `errno` to indicate the error.

Errors The `getrlimit()` and `setrlimit()` functions will fail if:

`EFAULT` The *rlp* argument 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 {`PRIV_SYS_RESOURCE`} is not asserted in the effective set of the current process.

The `setrlimit()` function may fail if:

`EINVAL` The limit specified cannot be lowered because current usage is already higher than the limit.

Usage The `getrlimit()` and `setrlimit()` functions have transitional interfaces for 64-bit file offsets. See `lf64(5)`.

The rlimit functionality is now provided by the more general resource control facility described on the `setrctl(2)` manual page. The actions associated with the resource limits described above are true at system boot, but an administrator can modify the local configuration to modify signal delivery or type. Application authors that utilize rlimits for the purposes of resource awareness should investigate the resource controls facility.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also rctladm(1M), brk(2), exec(2), fork(2), open(2), setrctl(2), sigaltstack(2), ulimit(2), getdtablesize(3C), malloc(3C), signal(3C), signal.h(3HEAD), sysconf(3C), attributes(5), lf64(5), privileges(5), resource_controls(5), standards(5)

Name getsid – get process group ID of session leader

Synopsis

```
#include <unistd.h>

pid_t getsid(pid_t pid);
```

Description The `getsid()` function obtains the process group ID of the process that is the session leader of the process specified by *pid*. If *pid* is `(pid_t) 0`, it specifies the calling process.

Return Values Upon successful completion, `getsid()` returns the process group ID of the session leader of the specified process. Otherwise, it returns `(pid_t)−1` and sets `errno` to indicate the error.

Errors The `getsid()` function will fail if:

`EPERM` The process specified by *pid* is not in the same session as the calling process, and the implementation does not allow access to the process group ID of the session leader of that process from the calling process.

`ESRCH` There is no process with a process ID equal to *pid*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also `exec(2)`, `fork(2)`, `getpid(2)`, `getpgid(2)`, `setpgid(2)`, `setsid(2)`, `attributes(5)`, `standards(5)`

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);
```

Description The `getuid()` function returns the real user ID of the calling process. The real user ID identifies the person who is logged in.

The `geteuid()` function 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.

The `getgid()` function returns the real group ID of the calling process.

The `getegid()` function returns the effective group ID of the calling process.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `Intro(2)`, `setuid(2)`, `attributes(5)`, `standards(5)`

Name getustack, setustack – retrieve or change the address of per-LWP stack boundary information

Synopsis

```
#include <ucontext.h>

int getustack(stack_t **spp);

int setustack(stack_t *sp);
```

Description The `getustack()` function retrieves the address of per-LWP stack boundary information. The address is stored at the location pointed to by *spp*. If this address has not been defined using a previous call to `setustack()`, `NULL` is stored at the location pointed to by *spp*.

The `setustack()` function changes the address of the current thread's stack boundary information to the value of *sp*.

Return Values Upon successful completion, these functions return 0. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors These functions will fail if:

`EFAULT` The *spp* or *sp* argument does not refer to a valid address.

Usage Only implementors of custom threading libraries should use these functions to get and set the address of the stack bound to an internal per-thread data structure. Other users should use `stack_getbounds(3C)` and `stack_setbounds(3C)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Async-Signal-Safe

See Also `_stack_grow(3C)`, `stack_getbounds(3C)`, `stack_inbounds(3C)`, `stack_setbounds(3C)`, `stack_violation(3C)`, `attributes(5)`

Name ioctl – control device

Synopsis

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

int ioctl(int fildes, int request, /* arg */ ...);
```

Description The `ioctl()` function performs a variety of control functions on devices and streams. For non-streams files, the functions performed by this call are device-specific control functions. The *request* argument 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()` function as described in `streamio(7I)`.

The *fildes* argument is an open file descriptor that refers to a device. The *request* argument selects the control function to be performed and depends on the device being addressed. The *arg* argument 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, `−1` is returned and `errno` is set to indicate the error.

Errors The `ioctl()` function will fail for any type of file if:

`EBADF` The *fildes* argument is not a valid open file descriptor.

`EINTR` A signal was caught during the execution of the `ioctl()` function.

`EINVAL` The stream or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

The `ioctl()` function will also fail if the device driver detects an error. In this case, the error is passed through `ioctl()` without change to the caller. A particular driver might not have all of the following error cases. Under the following conditions, requests to device drivers may fail and set `errno` to indicate the error

`EFAULT` The *request* argument requires a data transfer to or from a buffer pointed to by *arg*, but *arg* points to an illegal address.

`EINVAL` The *request* or *arg* argument is not valid for this device.

`EIO` Some physical I/O error has occurred.

`ENOLINK` The *fildes* argument is on a remote machine and the link to that machine is no longer active.

ENOTTY The *fildes* argument is not associated with a streams device that accepts control functions.

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

ENODEV The *fildes* argument refers to a valid streams device, but the corresponding device driver does not support the `ioctl()` function.

Streams errors are described in `streamio(7I)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also `attributes(5)`, `standards(5)`, `streamio(7I)`, `termio(7I)`

Name issetugid – determine if current executable is running setuid or setgid

Synopsis

```
#include <unistd.h>

int issetugid(void);
```

Description The `issetugid()` function enables library functions (in `libtermlib`, `libc`, or other libraries) to guarantee safe behavior when used in `setuid` or `setgid` programs or programs that run with more privileges after a succesful `exec(2)`. Some library functions might be passed insufficient information and not know whether the current program was started `setuid` or `setgid` because a higher level calling code might have made changes to the `uid`, `euid`, `gid`, or `egid`. These low-level library functions are therefore unable to determine if they are being run with elevated or normal privileges.

The `issetugid()` function should be used to determine if a path name returned from a `getenv(3C)` call can be used safely to open the specified file. It is often not safe to open such a file because the status of the effective `uid` is not known.

The result of a call to `issetugid()` is unaffected by calls to `setuid()`, `setgid()`, or other such calls. In case of a call to `fork(2)`, the child process inherits the same status.

The status of `issetugid()` is affected only by `execve()` (see `exec(2)`). If a child process executes a new executable file, a new `issetugid()` status will be based on the existing process's `uid`, `euid`, `gid`, and `egid` permissions and on the modes of the executable file. If the new executable file modes are `setuid` or `setgid`, or if the existing process is executing the new image with `uid` != `euid` or `gid` != `egid`, or if the permitted set before the call to the `exec` function is not a superset of the inheritable set at that time, `issetugid()` returns 1 in the new process.

Return Values The `issetugid()` function returns 1 if the process was made `setuid` or `setgid` as the result of the last or a previous call to `execve()`. Otherwise it returns 0.

Errors The `issetugid()` function is always successful. No return value is reserved to indicate an error.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Async-Signal-Safe

See Also `exec(2)`, `fork(2)`, `setuid(2)`, `getenv(3C)`, `attributes(5)`, `privileges(5)`

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);
```

Description The `kill()` function 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.h(3HEAD)`), 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 one of functions in the `exec(2)` family) user ID of the receiving process, unless the privilege {`PRIV_PROC_OWNER`} is asserted in the effective set of the sending process (see `Intro(2)`), or *sig* is `SIGCONT` and the sending process has the same session ID as the receiving process. A process needs the basic privilege {`PRIV_PROC_SESSION`} to send signals to a process with a different session ID. See `privileges(5)`.

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 {`PRIV_PROC_OWNER`} privilege is not asserted in the effective set of the sending process, *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 {`PRIV_PROC_OWNER`} privilege is asserted in the effective set of the sending process, *sig* will be sent to all processes excluding special processes.

Return Values Upon successful completion, 0 is returned. Otherwise, −1 is returned, no signal is sent, and `errno` is set to indicate the error.

Errors The `kill()` function will fail if:

`EINVAL` The *sig* argument is not a valid signal number.

`EPERM` The *sig* argument is `SIGKILL` and the *pid* argument 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*).

The effective user of the calling process does not match the real or saved user and the calling process does not have the {`PRIV_PROC_OWNER`} privilege asserted in the effective set, and the calling process either is not sending `SIGCONT` to a process that

shares the same session ID or does not have the {PRIV_PROC_SESSION} privilege asserted and is trying to send a signal to a process with a different session ID.

ESRCH No process or process group can be found corresponding to that specified by *pid*.

Usage The sigsend(2) function provides a more versatile way to send signals to processes.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also kill(1), Intro(2), exec(2), getpid(2), getsid(2), setpgrp(2), sigaction(2), sigsend(2), signal(3C), signal.h(3HEAD), attributes(5), privileges(5), standards(5)

Name link – link to a file

Synopsis

```
#include <unistd.h>

int link(const char *existing, const char *new);
```

Description The `link()` function creates a new link (directory entry) for the existing file and increments its link count by one. The *existing* argument points to a path name naming an existing file. The *new* argument points to a pathname 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. Privileged processes can make multiple links to a directory. Unless the caller is privileged, 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, `0` is returned. Otherwise, `−1` is returned, no link is created, and `errno` is set to indicate the error.

Errors The `link()` function will fail if:

`EACCES`	A component of either path prefix denies search permission, or the requested link requires writing in a directory with a mode that denies write permission.
`EDQUOT`	The directory where the entry for the new link is being placed cannot be extended because the user's quota of disk blocks on that file system has been exhausted.
`EEXIST`	The link named by *new* exists.
`EFAULT`	The *existing* or *new* argument points to an illegal address.
`EILSEQ`	The path argument includes non-UTF8 characters and the file system accepts only file names where all characters are part of the UTF-8 character codeset.
`EINTR`	A signal was caught during the execution of 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.
`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`	The *existing* or *new* argument is a null pathname; a component of either path prefix does not exist; or the file named by *existing* does not exist.

ENOLINK	The *existing* or *new* argument 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 {PRIV_SYS_LINKDIR} privilege is not asserted in the effective set of the calling process. The effective user ID does not match the owner of the file and the {PRIV_FILE_LINK_ANY} privilege is not asserted in the effective set of the calling process.
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).

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also symlink(2), unlink(2), attributes(5), privileges(5), standards(5)

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 The `llseek()` function 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*.
- If *whence* is `SEEK_HOLE`, the offset of the start of the next hole greater than or equal to the supplied offset is returned. The definition of a hole immediately follows this list.
- If *whence* is `SEEK_DATA`, the file pointer is set to the start of the next non-hole file region greater than or equal to the supplied offset.

A "hole" is defined as a contiguous range of bytes in a file, all having the value of zero, but not all zeros in a file are guaranteed to be represented as holes returned with `SEEK_HOLE`. Filesystems are allowed to expose ranges of zeros with `SEEK_HOLE`, but not required to. Applications can use `SEEK_HOLE` to optimise their behavior for ranges of zeros, but must not depend on it to find all such ranges in a file. The existence of a hole at the end of every data region allows for easy programming and implies that a virtual hole exists at the end of the file.

For filesystems that do not supply information about holes, the file will be represented as one entire data region.

Although each file has a 64-bit file pointer associated with it, some existing file system types (such as `tmpfs`) do not support the full range of 64-bit offsets. In particular, on such file systems, 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.

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

Return Values Upon successful completion, `llseek()` returns the resulting pointer location as measured in bytes from the beginning of the file. Remote file descriptors are the only ones that allow negative file pointers. Otherwise, `−1` is returned, the file pointer remains unchanged, and `errno` is set to indicate the error.

Errors The `llseek()` function will fail if:

`EBADF` The *fildes* argument is not an open file descriptor.

`EINVAL` The *whence* argument is not `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`; the *offset* argument is not a valid offset for this file system type; or the *fildes* argument is not a remote file descriptor and the resulting file pointer would be negative.

ENXIO	For SEEK_DATA, there are no more data regions past the supplied offset. For SEEK_HOLE, there are no more holes past the supplied offset.
ESPIPE	The *fildes* argument is associated with a pipe or FIFO.

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);
```

Description The `lseek()` function 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*.
- If *whence* is `SEEK_HOLE`, the offset of the start of the next hole greater than or equal to the supplied offset is returned. The definition of a hole is provided near the end of the DESCRIPTION.
- If *whence* is `SEEK_DATA`, the file pointer is set to the start of the next non-hole file region greater than or equal to the supplied offset.

The symbolic constants `SEEK_SET`, `SEEK_CUR`, `SEEK_END`, `SEEK_HOLE`, and `SEEK_DATA` are defined in the header `<unistd.h>`.

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

The `lseek()` function 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.

If *fildes* is a remote file descriptor and *offset* is negative, `lseek()` returns the file pointer even if it is negative. The `lseek()` function will not, by itself, extend the size of a file.

If *fildes* refers to a shared memory object, `lseek()` behaves as if *fildes* referred to a regular file.

A “hole” is defined as a contiguous range of bytes in a file, all having the value of zero, but not all zeros in a file are guaranteed to be represented as holes returned with `SEEK_HOLE`. Filesystems are allowed to expose ranges of zeros with `SEEK_HOLE`, but not required to. Applications can use `SEEK_HOLE` to optimise their behavior for ranges of zeros, but must not depend on it to find all such ranges in a file. The existence of a hole at the end of every data region allows for easy programming and implies that a virtual hole exists at the end of the file. Applications should use `fpathconf(_PC_MIN_HOLE_SIZE)` or `pathconf(_PC_MIN_HOLE_SIZE)` to determine if a filesystem supports `SEEK_HOLE`. See `fpathconf(2)`.

For filesystems that do not supply information about holes, the file will be represented as one entire data region.

Return Values Upon successful completion, the resulting offset, as measured in bytes from the beginning of the file, is returned. Otherwise, `(off_t)-1` is returned, the file offset remains unchanged, and `errno` is set to indicate the error.

Errors The `lseek()` function will fail if:

`EBADF` The *fildes* argument is not an open file descriptor.

`EINVAL` The *whence* argument is not `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`; or the *fildes* argument is not a remote file descriptor and the resulting file pointer would be negative.

`ENXIO` For `SEEK_DATA`, there are no more data regions past the supplied offset. For `SEEK_HOLE`, there are no more holes past the supplied offset.

`EOVERFLOW` The resulting file offset would be a value which cannot be represented correctly in an object of type `off_t` for regular files.

`ESPIPE` The *fildes* argument is associated with a pipe, a FIFO, or a socket.

Usage The `lseek()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

In multithreaded applications, using `lseek()` in conjunction with a `read(2)` or `write(2)` call on a file descriptor shared by more than one thread is not an atomic operation. To ensure atomicity, use `pread()` or `pwrite()`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `creat(2)`, `dup(2)`, `fcntl(2)`, `fpathconf(2)`, `open(2)`, `read(2)`, `write(2)`, `attributes(5)`, `lf64(5)`, `standards(5)`

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

The `_lwp_cond_broadcast()` function 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 signaled. 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 Upon successful completion, `0` is returned. A non-zero value indicates an error.

Errors The `_lwp_cond_signal()` and `_lwp_cond_broadcast()` functions will fail if:

`EINVAL` The *cvp* argument points to an invalid LWP condition variable.

`EFAULT` The *cvp* argument points to an invalid address.

See Also `_lwp_cond_wait(2)`, `_lwp_mutex_lock(2)`

Name _lwp_cond_wait, _lwp_cond_timedwait, _lwp_cond_reltimedwait – 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);

int _lwp_cond_reltimedwait(lwp_cond_t *cvp, lwp_mutex_t *mp,
     timestruc_t *reltime);
```

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 0 before use.

The `_lwp_cond_wait()` function 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.

The `_lwp_cond_timedwait()` function 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*, `_lwp_cond_timedwait()` returns with the error code `ETIME`.

The `_lwp_cond_reltimedwait()` function is similar to `_lwp_cond_wait()`, except that the calling LWP will not block past the relative time specified by *reltime*. If the time of day becomes greater than the starting time of day plus *reltime*, `_lwp_cond_reltimedwait()` returns with the error code `ETIME`.

The `_lwp_cond_wait()`, `_lwp_cond_timedwait()`, and `_lwp_cond_reltimedwait()` functions always return with the mutex locked and owned by the calling lightweight process.

Return Values Upon successful completion, `0` is returned. A non-zero value indicates an error.

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

`EINVAL` The *cvp* argument points to an invalid LWP condition variable or the *mp* argument points to an invalid LWP mutex.

`EFAULT` The *mp*, *cvp*, or *abstime* argument points to an illegal address.

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

`ETIME` The time specified in*abstime* or *reltime* has passed.

Examples

EXAMPLE 1 Use the `_lwp_cond_wait()` function in a loop testing some condition.

The `_lwp_cond_wait()` function 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);
```

EXAMPLE 2 Use the `_lwp_cond_timedwait()` function in a loop testing some condition.

The `_lwp_cond_timedwait()` function 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;
        SENDwhom}
}
(void) _lwp_mutex_unlock(&m);
```

This example 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.

EXAMPLE 3 Use the `_lwp_cond_reltimedwait()` function in a loop testing some condition.

The `_lwp_cond_reltimedwait()` function is also normally used in a loop testing some condition. It uses a relative timeout value as follows:

EXAMPLE 3 Use the `_lwp_cond_reltimedwait()` function in a loop testing some condition. *(Continued)*

```
timestruc_t to;
lwp_mutex_t m;
lwp_cond_t cv;
int cond, err;
(void) _lwp_mutex_lock(&m);
while (cond == FALSE) {
        to.tv_sec = TIMEOUT;
        to.tv_nsec = 0;
        err = _lwp_cond_reltimedwait(&cv, &m, &to);
        if (err == ETIME) {
                /* timeout, do something */
                break;
        }
}
(void) _lwp_mutex_unlock(&m);
```

See Also

`_lwp_cond_broadcast(2)`, `_lwp_cond_signal(2)`, `_lwp_kill(2)`, `_lwp_mutex_lock(2)`, `fork(2)`, `kill(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 The `_lwp_info()` function 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;
```

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

The `lwp_stime` member 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` The *buffer* argument points to an illegal address.

Additionally, the `_lwp_info()` function will fail for 32-bit interfaces if:

`EOVERFLOW` The size of the `tv_sec` member of the `timestruc_t` type pointed to by `lwp_utime` and `lwp_stime` is too small to contain the correct number of seconds.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

See Also `times(2)`, `attributes(5)`

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 The `_lwp_kill()` function 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.h`(3HEAD). 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 Upon successful completion, `0` is returned. A non-zero value indicates an error.

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

`EINVAL` The *sig* argument is not a valid signal number.

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

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

See Also `kill`(2), `sigaction`(2), `sigprocmask`(2), `signal.h`(3HEAD), `attributes`(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 0 before use.

The `_lwp_mutex_lock()` function 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".

The `_lwp_mutex_trylock()` function 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.

The `_lwp_mutex_unlock()` function 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 Upon successful completion, `0` is returned. 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` The *mp* argument points to an invalid LWP mutex.

`EFAULT` The *mp* argument points to an illegal address.

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

`EBUSY` The *mp* argument 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 The `_lwp_self()` function returns the ID of the calling LWP.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

See Also `attributes(5)`

Name _lwp_sema_wait, _lwp_sema_trywait, _lwp_sema_init, _lwp_sema_post – semaphore operations

Synopsis

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

int _lwp_sema_wait(lwp_sema_t *sema);

int _lwp_sema_trywait(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. The `_lwp_sema_init()` function initializes the count, `_lwp_sema_post()` atomically increments the count, and `_lwp_sema_wait()` waits for the count to become greater than 0 and then atomically decrements it.

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

The `_lwp_sema_wait()` function blocks the calling LWP until the semaphore count becomes greater than 0 and then atomically decrements it.

The `_lwp_sema_trywait()` function atomically decrements the count if it is greater than zero. Otherwise it returns an error.

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

Return Values Upon successful completion, `0` is returned. A non-zero value indicates an error.

Errors The `_lwp_sema_init()`, `_lwp_sema_trywait()`, `_lwp_sema_wait()`, and `_lwp_sema_post()` functions will fail if:

`EINVAL` The *sema* argument points to an invalid semaphore.

`EFAULT` The *sema* argument points to an illegal address.

The `_lwp_sema_wait()` function will fail if:

`EINTR` The function execution was interrupted by a signal or `fork(2)`.

The `_lwp_sema_trywait()` function will fail if:

`EBUSY` The function was called on a semaphore with a zero count.

The `_lwp_sema_post()` function will fail if:

`EOVERFLOW` The value of the *sema* argument exceeds `SEM_VALUE_MAX`.

See Also `fork(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 The `_lwp_suspend()` function 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.

The `_lwp_continue()` function 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 Upon successful completion, `0` is returned. 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` The *target_lwpid* argument cannot be found in the current process.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

See Also `attributes(5)`

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);
```

Description The `memcntl()` function 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*).

The *addr* argument 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. The selection criteria are constructed by a bitwise OR operation on the attribute bits and must match exactly.

`PROT_READ` Page can be read.

`PROT_WRITE` Page can be written.

`PROT_EXEC` Page can be executed.

The following criteria may also be specified:

`PROC_TEXT` Process text.

`PROC_DATA` Process data.

The `PROC_TEXT` attribute specifies all privately mapped segments with read and execute permission, and the `PROC_DATA` attribute 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.

The *arg* argument is not used, 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*. The *addr* and *len* arguments are not used, but must be `NULL` and `0` respectively, to ensure compatibility with potential future enhancements. The *arg* argument 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. The *arg* argument 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` Return immediately once all write operations are scheduled; with `MS_SYNC` the function will not return until all write operations are completed.

`MS_INVALIDATE` Invalidate 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*. The *arg* argument is not used, 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*. The *addr*, *len*, and *arg* arguments are not used, but must be NULL, 0 and 0, respectively, to ensure compatibility with potential future enhancements.

MC_HAT_ADVISE
: Advise system how a region of user-mapped memory will be accessed. The *arg* argument is interpreted as a "struct memcntl_mha *". The following members are defined in a struct memcntl_mha:

```
uint_t mha_cmd;
uint_t mha_flags;
size_t mha_pagesize;
```

The accepted values for mha_cmd are:

```
MHA_MAPSIZE_VA
MHA_MAPSIZE_STACK
MHA_MAPSIZE_BSSBRK
```

The mha_flags member is reserved for future use and must always be set to 0. The mha_pagesize member must be a valid size as obtained from getpagesizes(3C) or the constant value 0 to allow the system to choose an appropriate hardware address translation mapping size.

MHA_MAPSIZE_VA sets the preferred hardware address translation mapping size of the region of memory from *addr* to *addr* + *len*. Both *addr* and *len* must be aligned to an mha_pagesize boundary. The entire virtual address region from *addr* to *addr* + *len* must not have any holes. Permissions within each mha_pagesize–aligned portion of the region must be consistent. When a size of 0 is specified, the system selects an appropriate size based on the size and alignment of the memory region, type of processor, and other considerations.

MHA_MAPSIZE_STACK sets the preferred hardware address translation mapping size of the process main thread stack segment. The *addr* and *len* arguments must be NULL and 0, respectively.

MHA_MAPSIZE_BSSBRK sets the preferred hardware address translation mapping size of the process heap. The *addr* and *len* arguments must be NULL and 0, respectively. See the NOTES section of the ppgsz(1) manual page for additional information on process heap alignment.

The *attr* argument must be 0 for all MC_HAT_ADVISE operations.

The *mask* argument must be 0; it is reserved for future use.

Locks established with the lock operations are not inherited by a child process after fork(2). The memcntl() function fails if it attempts to lock more memory than a system-specific limit.

Due to the potential impact on system resources, the operations MC_LOCKAS, MC_LOCK, MC_UNLOCKAS, and MC_UNLOCK are restricted to privileged processes.

Usage The memcntl() function subsumes the operations of plock(3C).

MC_HAT_ADVISE is intended to improve performance of applications that use large amounts of memory on processors that support multiple hardware address translation mapping sizes; however, it should be used with care. Not all processors support all sizes with equal efficiency. Use of larger sizes may also introduce extra overhead that could reduce performance or available memory. Using large sizes for one application may reduce available resources for other applications and result in slower system wide performance.

Return Values Upon successful completion, memcntl() returns 0; otherwise, it returns −1 and sets errno to indicate an error.

Errors The memcntl() function will fail if:

EAGAIN When the selection criteria match, some or all of the memory identified by the operation could not be locked when MC_LOCK or MC_LOCKAS was specified, some or all mappings in the address range [*addr*, *addr* + *len*) are locked for I/O when MC_HAT_ADVISE was specified, or the system has insufficient resources when MC_HAT_ADVISE was specified.

The *cmd* is MC_LOCK or MC_LOCKAS and locking the memory identified by this operation would exceed a limit or resource control on locked memory.

EBUSY When the selection criteria match, some or all of the addresses in the range [*addr, addr* + *len*) are locked and MC_SYNC with the MS_INVALIDATE option was specified.

EINVAL The *addr* argument specifies invalid selection criteria or is not a multiple of the page size as returned by sysconf(3C); the *addr* and/or *len* argument does not have the value 0 when MC_LOCKAS or MC_UNLOCKAS is specified; the *arg* argument is not valid for the function specified; mha_pagesize or mha_cmd is invalid; or MC_HAT_ADVISE is specified and not all pages in the specified region have the same access permissions within the given size boundaries.

ENOMEM When the selection criteria match, some or all of the addresses in the range [*addr*, *addr* + *len*) are invalid for the address space of a process or specify one or more pages which are not mapped.

EPERM The {PRIV_PROC_LOCK_MEMORY} privilege is not asserted in the effective set of the calling process and MC_LOCK, MC_LOCKAS, MC_UNLOCK, or MC_UNLOCKAS was specified.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also ppgsz(1), fork(2), mmap(2), mprotect(2), getpagesizes(3C), mlock(3C), mlockall(3C), msync(3C), plock(3C), sysconf(3C), attributes(5), privileges(5)

Name meminfo – provide information about memory

Synopsis

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

int meminfo(const uint64_t inaddr[], int addr_count,
     const uint_t info_req[], int info_count, uint64_t outdata[],
     uint_t validity[]);
```

Parameters

inaddr	array of input addresses; the maximum number of addresses that can be processed for each call is `MAX_MEMINFO_CNT`
addr_count	number of addresses
info_req	array of types of information requested
info_count	number of pieces of information requested for each address in *inaddr*
outdata	array into which results are placed; array size must be the product of *info_count* and *addr_count*
validity	array of size *addr_count* containing bitwise result codes; 0th bit evaluates validity of corresponding input address, 1st bit validity of response to first member of *info_req*, and so on

Description The `meminfo()` function provides information about virtual and physical memory particular to the calling process. The user or developer of performance utilities can use this information to analyze system memory allocations and develop a better understanding of the factors affecting application performance.

The caller of `meminfo()` can obtain the following types of information about both virtual and physical memory.

`MEMINFO_VPHYSICAL`	physical address corresponding to virtual address
`MEMINFO_VLGRP`	locality group of physical page corresponding to virtual address
`MEMINFO_VPAGESIZE`	size of physical page corresponding to virtual address
`MEMINFO_VREPLCNT`	number of replicated physical pages corresponding to specified virtual address
`MEMINFO_VREPL` | *n*	*n*th physical replica of specified virtual address
`MEMINFO_VREPL_LGRP` | *n*	lgrp of *n*th physical replica of specified virtual address
`MEMINFO_PLGRP`	locality group of specified physical address

Return Values Upon successful completion `meminfo()` returns 0. Otherwise −1 is returned and `errno` is set to indicate the error.

Errors The `meminfo()` function will fail if:

`EFAULT` The area pointed to by *outdata* or *validity* could not be written, or the data pointed to by *info_req* or *inaddr* could not be read.

`EINVAL` The value of *info_count* is greater than 31 or less than 1, or the value of *addr_count* is less than 1.

Examples **EXAMPLE 1** Print physical pages and page sizes corresponding to a set of virtual addresses.

The following example prints the physical pages and page sizes corresponding to a set of virtual addresses.

```
void
print_info(void **addrvec, int how_many)
{
    static const uint_t info[] = {
        MEMINFO_VPHYSICAL,
        MEMINFO_VPAGESIZE
    };

    int info_num = sizeof (info) / sizeof (info[0]);
    int i;

    uint64_t *inaddr = alloca(sizeof (uint64_t) * how_many);
    uint64_t *outdata = alloca(sizeof (uint64_t) * how_many * info_num);
    uint_t *validity = alloca(sizeof (uint_t) * how_many);

    for (i = 0; i < how_many; i++)
        inaddr[i] = (uint64_t)addrvec[i];

    if (meminfo(inaddr, how_many, info, info_num, outdata,
                validity) < 0) {
        perror("meminfo");
        return;
    }

    for (i = 0; i < how_many; i++) {
        if ((validity[i] & 1) == 0)
            printf("address 0x%llx not part of address space\n",
                inaddr[i]);

        else if ((validity[i] & 2) == 0)
            printf("address 0x%llx has no physical page "
                "associated with it\n", inaddr[i]);

        else {
            char buff[80];
```

EXAMPLE 1 Print physical pages and page sizes corresponding to a set of virtual addresses. *(Continued)*

```
            if ((validity[i] & 4) == 0)
                strcpy(buff, "<Unknown>");
            else
                sprintf(buff, "%lld",
                    outdata[i * info_num + 1]);

            printf("address 0x%llx is backed by physical "
                "page 0x%llx of size %s\n",
                inaddr[i], outdata[i * info_num], buff);
        }
    }
}
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Stable
MT-Level	Async-Signal-Safe

See Also

memcntl(2), mmap(2), gethomelgroup(3C), getpagesize(3C), madvise(3C), sysconf(3C), attributes(5)

Name mincore – determine residency of memory pages

Synopsis

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

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

Description The `mincore()` function 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, and to 0 to indicate that 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 between the time `mincore()` checks and returns the information, returned information might be outdated. Only locked pages are guaranteed to remain in memory; see `mlock(3C)`.

Return Values Upon successful completion, `mincore()` returns `0`. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `mincore()` function will fail if:

`EFAULT` The *vec* argument points to an illegal address.

`EINVAL` The *addr* argument is not a multiple of the page size as returned by `sysconf(3C)`, or 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.

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);
```

Description The `mkdir()` function 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* names a symbolic link, `mkdir()` fails and sets errno to `EEXIST`.

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, `0` is returned. Otherwise, `−1` is returned, no directory is created, and `errno` is set to indicate the error.

Errors The `mkdir()` function will fail if:

`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.
`EDQUOT`	The directory where the new file entry is being placed cannot be extended because the user's quota of disk blocks on that file system has been exhausted; the new directory cannot be created because the user's quota of disk blocks on that file system has been exhausted; or the user's quota of inodes on the file system where the file is being created has been exhausted.
`EEXIST`	The named file already exists.
`EFAULT`	The *path* argument points to an illegal address.
`EINVAL`	An attempt was made to create an extended attribute that is a directory.
`EIO`	An I/O error has occurred while accessing the file system.

EILSEQ	The path argument includes non-UTF8 characters and the file system accepts only file names where all characters are part of the UTF-8 character codeset.
ELOOP	Too many symbolic links were encountered in translating *path*, or a loop exists in symbolic links encountered during resolution of *path*
EMLINK	The maximum number of links to the parent directory would be exceeded.
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	The *path* argument 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.

The mkdir() function may fail if:

ENAMETOOLONG	As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

Examples

EXAMPLE 1 Create a directory.

The following example demonstrates how to create a directory named /home/cnd/mod1, with read, write, and search permissions for owner and group, and with read and search permissions for others.

```
#include <sys/types.h>
#include <sys/stat.h>
int status;
...
status = mkdir("/home/cnd/mod1",
     S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also chmod(2), mknod(2), umask(2), mkdirp(3GEN), stat.h(3HEAD), attributes(5), standards(5)

Name mknod – make a directory, a special file, or a regular file

Synopsis

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

int mknod(const char *path, mode_t mode, dev_t dev);
```

Description The `mknod()` function 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 a bitwise OR operation 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	On directories, restricted deletion flag; on regular files on a UFS file system, do not cache flag.
`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).

If *path* is a symbolic link, it is not followed.

Return Values Upon successful completion, mknod() returns 0. Otherwise, it returns −1, the new file is not created, and errno is set to indicate the error.

Errors The mknod() function will fail if:

EACCES	A component of the path prefix denies search permission, or write permission is denied on the parent directory.
EDQUOT	The directory where the new file entry is being placed cannot be extended because the user's quota of disk blocks on that file system has been exhausted, or the user's quota of inodes on the file system where the file is being created has been exhausted.
EEXIST	The named file exists.
EFAULT	The *path* argument points to an illegal address.
EINTR	A signal was caught during the execution of the mknod() function.
EINVAL	An invalid argument exists.
EIO	An I/O error occurred while accessing 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	A component of the path prefix specified by *path* does not name an existing directory or *path* is an empty string.
ENOLINK	The *path* argument points to a remote machine and the link to that machine is no longer active.

ENOSPC	The directory that would contain the new file cannot be extended or the file system is out of file allocation resources.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	Not all privileges are asserted in the effective set of the calling process.
EROFS	The directory in which the file is to be created is located on a read-only file system.

The mknod() function may fail if:

ENAMETOOLONG	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

Usage Applications should use the mkdir(2) function to create a directory because appropriate permissions are not required and because mknod() might not establish directory entries for the directory itself (.) and the parent directory (. .). The mknod() function can be invoked only by a privileged user for file types other than FIFO special. The mkfifo(3C) function should be used to create FIFOs.

Doors are created using door_create(3C) and can be attached to the file system using fattach(3C). Symbolic links can be created using symlink(2). An endpoint for communication can be created using socket(3SOCKET).

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also chmod(2), creat(2), exec(2), mkdir(2), open(2), stat(2), symlink(2), umask(2), door_create(3C), fattach(3C), makedev(3C), mkfifo(3C), socket(3SOCKET), stat.h(3HEAD), attributes(5), privileges(5), standards(5)

Name mmap – map pages of memory

Synopsis

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

void *mmap(void *addr, size_t len, int prot, int flags,
     int fildes, off_t off);
```

Description The `mmap()` function establishes a mapping between a process's address space and a file or shared memory object. The format of the call is as follows:

pa = `mmap(`*addr*`,` *len*`,` *prot*`,` *flags*`,` *fildes*`,` *off*`);`

The `mmap()` function establishes a mapping between the address space of the process 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 a function of the *addr* argument and values of *flags*, further described below. A successful `mmap()` call returns *pa* as its result. The address range starting at *pa* and continuing for *len* bytes will be legitimate for the possible (not necessarily current) address space of the process. The range of bytes starting at *off* and continuing for *len* bytes will be legitimate for the possible (not necessarily current) offsets in the file or shared memory object represented by *fildes*.

The `mmap()` function 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 is created prior to the `mmap()` call and has no contents, or when the file is truncated. Any reference to addresses beyond the end of the object, however, will result in the delivery of a `SIGBUS` or `SIGSEGV` signal. The `mmap()` function cannot be used to implicitly extend the length of files.

The mapping established by `mmap()` replaces any previous mappings for those whole pages containing any part of the address space of the process starting at *pa* and continuing for *len* bytes.

If the size of the mapped file changes after the call to `mmap()` as a result of some other operation on the mapped file, the effect of references to portions of the mapped region that correspond to added or removed portions of the file is unspecified.

The `mmap()` function is supported for regular files and shared memory objects. Support for any other type of file is unspecified.

The *prot* argument determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped. The *prot* argument should be either `PROT_NONE` or the bitwise inclusive OR of one or more of the other flags in the following table, defined in the header <`sys/mman.h`>.

`PROT_READ` Data can be read.

`PROT_WRITE` Data can be written.

`PROT_EXEC` Data can be executed.

`PROT_NONE` Data cannot be accessed.

If an implementation of `mmap()` for a specific platform cannot support the combination of access types specified by *prot*, the call to `mmap()` fails. An implementation may permit accesses other than those specified by *prot*; however, the implementation will not permit a write to succeed where `PROT_WRITE` has not been set or permit any access where `PROT_NONE` alone has been set. Each platform-specific implementation of `mmap()` supports the following values of *prot*: `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, and the inclusive OR of `PROT_READ` and `PROT_WRITE`. On some platforms, the `PROT_WRITE` protection option is implemented as `PROT_READ|PROT_WRITE` and `PROT_EXEC` as `PROT_READ|PROT_EXEC`. The file descriptor *fildes* is opened with read permission, regardless of the protection options specified. If `PROT_WRITE` is specified, the application must have opened the file descriptor *fildes* with write permission unless `MAP_PRIVATE` is specified in the *flags* argument as described below.

The *flags* argument provides other information about the handling of the mapped data. The value of *flags* is the bitwise inclusive OR of these options, defined in `<sys/mman.h>`:

`MAP_SHARED`	Changes are shared.
`MAP_PRIVATE`	Changes are private.
`MAP_FIXED`	Interpret *addr* exactly.
`MAP_NORESERVE`	Do not reserve swap space.
`MAP_ANON`	Map anonymous memory.
`MAP_ALIGN`	Interpret *addr* as required aligment.
`MAP_TEXT`	Map text.
`MAP_INITDATA`	Map initialized data segment.

The `MAP_SHARED` and `MAP_PRIVATE` options describe the disposition of write references to the underlying object. If `MAP_SHARED` is specified, write references will change the memory object. If `MAP_PRIVATE` is specified, the initial write reference will create a private copy of the memory object page and redirect the mapping to the copy. The private copy is not created until the first write; until then, other users who have the object mapped `MAP_SHARED` can change the object. Either `MAP_SHARED` or `MAP_PRIVATE` must be specified, but not both. The mapping type is retained across `fork(2)`.

When `MAP_FIXED` is set in the *flags* argument, the system is informed that the value of *pa* must be *addr*, exactly. If `MAP_FIXED` is set, `mmap()` may return `(void *)`−1 and set `errno` to `EINVAL`. If a `MAP_FIXED` request is successful, the mapping established by `mmap()` replaces any previous mappings for the process's pages in the range [*pa, pa* + *len*). The use of `MAP_FIXED` is discouraged, since it may prevent a system from making the most effective use of its resources.

When `MAP_FIXED` is set and the requested address is the same as previous mapping, the previous address is unmapped and the new mapping is created on top of the old one.

When MAP_FIXED is not set, the system uses *addr* to arrive at *pa*. The *pa* so chosen will be an area of the address space that the system deems suitable for a mapping of *len* bytes to the file. The mmap() function interprets an *addr* value of 0 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".

When MAP_ALIGN is set, the system is informed that the alignment of *pa* must be the same as *addr*. The alignment value in *addr* must be 0 or some power of two multiple of page size as returned by sysconf(3C). If addr is 0, the system will choose a suitable alignment.

The MAP_NORESERVE option specifies that no swap space be reserved for a mapping. Without this flag, the creation of a writable 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 or SIGSEGV signal is delivered to the writing process. MAP_NORESERVE mappings are inherited across fork(); at the time of the fork(), 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.

When MAP_ANON is set in *flags*, and *fildes* is set to -1, mmap() provides a direct path to return anonymous pages to the caller. This operation is equivalent to passing mmap() an open file descriptor on /dev/zero with MAP_ANON elided from the *flags* argument.

The MAP_TEXT option informs the system that the mapped region will be used primarily for executing instructions. This information can help the system better utilize MMU resources on some platforms. This flag is always passed by the dynamic linker when it maps text segments of shared objects. When the MAP_TEXT option is used for regular file mappings on some platforms, the system can choose a mapping size larger than the page size returned by sysconf(3C). The specific page sizes that are used depend on the platform and the alignment of the addr and len arguments. Several diffrent mapping sizes can be used to map the region with larger page sizes used in the parts of the region that meet alignment and size requirements for those page sizes.

The MAP_INITDATA option informs the system that the mapped region is an initialized data segment of an executable or shared object. When the MAP_INITDATA option is used for regular file mappings on some platforms, the system can choose a mapping size larger than the page size returned by sysconf(). The MAP_INITDATA option should be used only by the dynamic linker for mapping initialized data of shared objects.

The *off* argument is constrained to be aligned and sized according to the value returned by sysconf() when passed _SC_PAGESIZE or _SC_PAGE_SIZE. When MAP_FIXED is specified, the

addr argument must also meet these constraints. The system performs mapping operations over whole pages. Thus, while the *len* argument 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` or `SIGSEGV` signal. `SIGBUS` signals may also be delivered on various file system conditions, including quota exceeded errors.

The `mmap()` function adds an extra reference to the file associated with the file descriptor *fildes* which is not removed by a subsequent `close(2)` on that file descriptor. This reference is removed when there are no more mappings to the file by a call to the `munmap(2)` function.

The `st_atime` field of the mapped file may be marked for update at any time between the `mmap()` call and the corresponding `munmap(2)` call. The initial read or write reference to a mapped region will cause the file's `st_atime` field to be marked for update if it has not already been marked for update.

The `st_ctime` and `st_mtime` fields of a file that is mapped with `MAP_SHARED` and `PROT_WRITE`, will be marked for update at some point in the interval between a write reference to the mapped region and the next call to `msync(3C)` with `MS_ASYNC` or `MS_SYNC` for that portion of the file by any process. If there is no such call, these fields may be marked for update at any time after a write reference if the underlying file is modified as a result.

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`.

The `mmap()` function aligns based on the length of the mapping. When determining the amount of space to add to the address space, `mmap()` includes two 8-Kbyte pages, one at each end of the mapping that are not mapped and are therefore used as “red-zone” pages. Attempts to reference these pages result in access violations.

The size requested is incremented by the 16 Kbytes for these pages and is then subject to rounding constraints. The constraints are:

- For 32-bit processes:

```
If length > 4 Mbytes
        round to 4-Mbyte multiple
elseif length > 512 Kbytes
        round to 512-Kbyte multiple
else
        round to 64-Kbyte multiple
```

- For 64-bit processes:

```
If length > 4 Mbytes
        round to 4-Mbyte multiple
else
        round to 1-Mbyte multiple
```

The net result is that for a 32-bit process:

- If an `mmap()` request is made for 4 Mbytes, it results in 4 Mbytes + 16 Kbytes and is rounded up to 8 Mbytes.
- If an `mmap()` request is made for 512 Kbytes, it results in 512 Kbytes + 16 Kbytes and is rounded up to 1 Mbyte.
- If an `mmap()` request is made for 1 Mbyte, it results in 1 Mbyte + 16 Kbytes and is rounded up to 1.5 Mbytes.
- Each 8-Kbyte mmap request "consumes" 64 Kbytes of virtual address space.

To obtain maximal address space usage for a 32-bit process:

- Combine 8-Kbyte requests up to a limit of 48 Kbytes.
- Combine amounts over 48 Kbytes into 496-Kbyte chunks.
- Combine amounts over 496 Kbytes into 4080-Kbyte chunks.

To obtain maximal address space usage for a 64-bit process:

- Combine amounts < 1008 Kbytes into chunks <= 1008 Kbytes.
- Combine amounts over 1008 Kbytes into 4080-Kbyte chunks.

The following is the output from a 32-bit program demonstrating this:

map 8192 bytes: `0xff390000`
map 8192 bytes: `0xff380000` 64-Kbyte delta between starting addresses.

map 512 Kbytes: `0xff180000`
map 512 Kbytes: `0xff080000` 1–Mbyte delta between starting addresses.

map 496 Kbytes: `0xff000000`
map 496 Kbytes: `0xfef80000` 512-Kbyte delta between starting addresses

map 1 Mbyte: `0xfee00000`
map 1 Mbyte: `0xfec80000` 1536-Kbyte delta between starting addresses

map 1008 Kbytes: `0xfeb80000`
map 1008 Kbytes: `0xfea80000` 1-Mbyte delta between starting addresses

map 4 Mbytes: `0xfe400000`
map 4 Mbytes: `0xfdc00000` 8-Mbyte delta between starting addresses

map 4080 Kbytes: `0xfd800000`
map 4080 Kbytes: `0xfd400000` 4-Mbyte delta between starting addresses

The following is the output of the same program compiled as a 64-bit application:

```
map 8192 bytes: 0xffffffff7f000000
map 8192 bytes: 0xffffffff7ef00000      1-Mbyte delta between starting addresses

map 512 Kbytes: 0xffffffff7ee00000
map 512 Kbytes: 0xffffffff7ed00000      1-Mbyte delta between starting addresses

map 496 Kbytes: 0xffffffff7ec00000
map 496 Kbytes: 0xffffffff7eb00000      1-Mbyte delta between starting addresses

map 1 Mbyte: 0xffffffff7e900000
map 1 Mbyte: 0xffffffff7e700000         2-Mbyte delta between starting addresses

map 1008 Kbytes: 0xffffffff7e600000
map 1008 Kbytes: 0xffffffff7e500000     1–Mbyte delta between starting addresses

map 4 Mbytes: 0xffffffff7e000000
map 4 Mbytes: 0xffffffff7d800000        8-Mbyte delta between starting addresses

map 4080 Kbytes: 0xffffffff7d400000
map 4080 Kbytes: 0xffffffff7d000000     4-Mbyte delta between starting addresses
```

Return Values Upon successful completion, the `mmap()` function returns the address at which the mapping was placed (*pa*); otherwise, it returns a value of `MAP_FAILED` and sets `errno` to indicate the error. The symbol `MAP_FAILED` is defined in the header `<sys/mman.h>`. No successful return from `mmap()` will return the value `MAP_FAILED`.

If `mmap()` fails for reasons other than `EBADF`, `EINVAL` or `ENOTSUP`, some of the mappings in the address range starting at *addr* and continuing for *len* bytes may have been unmapped.

Errors The `mmap()` function will fail if:

`EACCES` The *fildes* file descriptor 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.

`EBADF` The *fildes* file descriptor is not open (and `MAP_ANON` was not specified).

`EINVAL` The arguments *addr* (if `MAP_FIXED` was specified) or *off* are not multiples of the page size as returned by `sysconf()`.

The argument *addr* (if `MAP_ALIGN` was specified) is not 0 or some power of two multiple of page size as returned by `sysconf(3C)`.

`MAP_FIXED` and `MAP_ALIGN` are both specified.

The field in *flags* is invalid (neither MAP_PRIVATE or MAP_SHARED is set).

The argument *len* has a value equal to 0.

MAP_ANON was specified, but the file descriptor was not −1.

MAP_TEXT was specified but PROT_EXEC was not.

MAP_TEXT and MAP_INITDATA were both specified.

EMFILE — The number of mapped regions would exceed an implementation-dependent limit (per process or per system).

ENODEV — The *fildes* argument refers to an object for which mmap() is meaningless, such as a terminal.

ENOMEM — The MAP_FIXED option was specified and the range [*addr, addr* + *len*) exceeds that allowed for the address space of a process.

The MAP_FIXED option was *not* specified and there is insufficient room in the address space to effect the mapping.

The mapping could not be locked in memory, if required by mlockall(3C), because it would require more space than the system is able to supply.

The composite size of *len* plus the lengths obtained from all previous calls to mmap() exceeds RLIMIT_VMEM (see getrlimit(2)).

ENOTSUP — The system does not support the combination of accesses requested in the *prot* argument.

ENXIO — Addresses in the range [*off, off* + *len*) are invalid for the object specified by *fildes*.

The MAP_FIXED option was specified in *flags* and the combination of *addr*, *len* and *off* is invalid for the object specified by *fildes*.

EOVERFLOW — The file is a regular file and the value of *off* plus *len* exceeds the offset maximum establish in the open file description associated with *fildes*.

The mmap() function may fail if:

EAGAIN — The file to be mapped is already locked using advisory or mandatory record locking. See fcntl(2).

Usage Use of mmap() may reduce the amount of memory available to other memory allocation functions.

MAP_ALIGN is useful to assure a properly aligned value of *pa* for subsequent use with memcntl(2) and the MC_HAT_ADVISE command. This is best used for large, long-lived, and heavily referenced regions. MAP_FIXED and MAP_ALIGN are always mutually-exclusive.

Use of MAP_FIXED may result in unspecified behavior in further use of brk(2), sbrk(2), malloc(3C), and shmat(2). The use of MAP_FIXED is discouraged, as it may prevent an implementation from making the most effective use of resources.

The application must ensure correct synchronization when using mmap() in conjunction with any other file access method, such as read(2) and write(2), standard input/output, and shmat(2).

The mmap() function has a transitional interface for 64-bit file offsets. See lf64(5).

The mmap() function 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, whence)
read(fildes, buf, len)
/* use data in buf */
```

The following 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 */
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also

close(2), exec(2), fcntl(2), fork(2), getrlimit(2), memcntl(2), mmapobj(2), mprotect(2), munmap(2), shmat(2), lockf(3C), mlockall(3C), msync(3C), plock(3C), sysconf(3C), attributes(5), lf64(5), standards(5), null(7D), zero(7D)

Name mmapobj – map a file object in the appropriate manner

Synopsis

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

int mmapobj(int fd, uint_t flags, mmapobj_result_t *storage,
     uint_t *elements, void *arg);
```

Parameters

fd The open file descriptor for the file to be mapped.

flags Indicates that the default behavior of `mmapobj()` should be modified accordingly. Available flags are:

`MMOBJ_INTERPRET`
Interpret the contents of the file descriptor instead of just mapping it as a single image. This flag can be used only with `ELF` and `AOUT` files.

`MMOBJ_PADDING`
When mapping in the file descriptor, add an additional mapping before the lowest mapping and after the highest mapping. The size of this padding is at least as large as the amount pointed to by *arg*. These mappings will be private to the process, will not reserve any swap space and will have no protections. To use this address space, the protections for it will need to be changed. This padding request will be ignored for the `AOUT` format.

storage A pointer to the `mmapobj_result_t` array where the mapping data will be copied out after a successful mapping of *fd*.

elements A pointer to the number of `mmapobj_result_t` elements pointed to by *storage*. On return, *elements* contains the number of mappings required to fully map the requested object. If the original value of *elements* is too small, `E2BIG` is returned and *elements* is modified to contain the number of mappings necessary.

arg A pointer to additional information that might be associated with the specific request. Only the `MMOBJ_PADDING` request uses this argument. If `MMOBJ_PADDING` is not specified, *arg* must be `NULL`.

Description The `mmapobj()` function establishes a set of mappings between a process's address space and a file. By default, `mmapobj()` maps the whole file as a single, private, read-only mapping. The `MMOBJ_INTERPRET` flag instructs `mmapobj()` to attempt to interpret the file and map the file according to the rules for that file format. The following `ELF` and `AOUT` formats are supported:

`ET_EXEC` and `AOUT` executables
This format results in one or more mappings whose size, alignment and protections are as described by the file's program header information. The address of each mapping is explicitly defined by the file's program headers.

`ET_DYN` and `AOUT` shared objects
This format results in one or more mappings whose size, alignment and protections are as described by the file's program header information. The base address of the initial mapping

is chosen by `mmapobj()`. The addresses of adjacent mappings are based off of this base address as defined by the file's program headers.

`ET_REL` and `ET_CORE`
: This format results in a single, read-only mapping that covers the whole file. The base address of this mapping is chosen by `mmapobj()`.

The `mmapobj()` function will not map over any currently used mappings within the process, except for the case of an `ELF ET_EXEC` file for which a previous reservation has been made via `/dev/null`. The most common way to make such a reservation would be with an `mmap()` of `/dev/null`.

Mappings created with `mmapobj()` can be processed individually by other system calls such as `munmap(2)`.

The `mmapobj_result` structure contains the following members:

```
typedef struct mmapobj_result {
      caddr_t         mr_addr;         /* mapping address */
      size_t          mr_msize;        /* mapping size */
      size_t          mr_fsize;        /* file size */
      size_t          mr_offset;       /* offset into file */
      uint_t          mr_prot;         /* the protections provided */
      uint_t          mr_flags;        /* info on the mapping */
} mmapobj_result_t;
```

The macro `MR_GET_TYPE`(*mr_flags*) must be used when looking for the above flags in the value of *mr_flags*.

Values for *mr_flags* include:

```
MR_PADDING   0x1  /* this mapping represents requested padding */
MR_HDR_ELF   0x2  /* the ELF header is mapped at mr_addr */
MR_HDR_AOU   0x3  /* the AOUT header is mapped at mr_addr */
```

When `MR_PADDING` is set, *mr_fsize* and *mr_offset* will both be 0.

The *mr_fsize* member represents the amount of the file that is mapped into memory with this mapping.

The *mr_offset* member is the offset into the mapping where valid data begins.

The *mr_msize* member represents the size of the memory mapping starting at *mr_addr*. This size may include unused data prior to *mr_offset* that exists to satisfy the alignment requirements of this segment. This size may also include any non-file data that are required to provide `NOBITS` data (typically `.bss`). The system reserves the right to map more than *mr_msize* bytes of memory but only *mr_msize* bytes will be available to the caller of `mmapobj()`.

Return Values Upon successful completion, 0 is returned and *elements* contains the number of program headers that are mapped for *fd*. The data describing these elements are copied to storage such that the first *elements* members of the storage array contain valid mapping data.

On failure, -1 is returned and `errno` is set to indicate the error. No data is copied to storage.

Errors The `mmapobj()` function will fail if:

`E2BIG` The *elements* argument was not large enough to hold the number of loadable segments in *fd*. The *elements* argument will be modified to contain the number of segments required.

`EACCES` The file system containing the *fd* to be mapped does not allow execute access, or the file descriptor pointed to by *fd* is not open for reading.

`EADDRINUSE` The mapping requirements overlap an object that is already used by the process.

`EAGAIN` There is 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` The *fd* argument is not a valid open file descriptor.

`EFAULT` The *storage*, *arg*, or *elements* argument points to an invalid address.

`EINVAL` The *flags* argument contains an invalid flag.

`MMOBJ_PADDING` was not specified in *flags*and *arg* was non-null.

`ENODEV` The *fd* argument refers to an object for which `mmapobj()` is meaningless, such as a terminal.

`ENOMEM` Insufficient memory is available to hold the program headers.

Insufficient memory is available in the address space to create the mapping.

`ENOTSUP` The current user data model does not match the *fd* to be interpreted; thus, a 32-bit process that tried to use `mmapobj()` to interpret a 64-bit object would return `ENOTSUP`.

The *fd* argument is a file whose type can not be interpreted and `MMOBJ_INTERPRET` was specified in *flags*.

The `ELF` header contains an unaligned *e_phentsize* value.

`ENOSYS` An unsupported filesystem operation was attempted while trying to map in the object.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Private
MT-Level	Async-Signal-Safe

See Also ld.so.1(1), fcntl(2), memcntl(2), mmap(2), mprotect(2), munmap(2), elf(3ELF), madvise(3C), mlockall(3C), msync(3C), a.out(4), attributes(5)

Linker and Libraries Guide

Name mount – mount a file system

Synopsis

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

int mount(const char *spec, const char *dir, int mflag,
     char *fstype, char *dataptr,int datalen, char *optptr,
     int optlen);
```

Description The `mount()` function requests that a removable file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. The *spec* and *dir* arguments are pointers to path names.

After a successful call to `mount()`, all references to the file *dir* refer to the root directory on the mounted file system. The mounted file system is inserted into the kernel list of all mounted file systems. This list can be examined through the mounted file system table (see `mnttab(4)`).

The *fstype* argument is the file system type name. Standard file system names are defined with the prefix `MNTTYPE_` in `<sys/mntent.h>`. If neither `MS_DATA` nor `MS_OPTIONSTR` is set in *mflag*, then *fstype* is ignored and the type of the root file system is assumed.

The *dataptr* argument is 0 if no file system-specific data is to be passed; otherwise it points to an area of size *datalen* that contains the file system-specific data for this mount and the `MS_DATA` flag should be set.

If the `MS_OPTIONSTR` flag is set, then *optptr* points to a buffer containing the list of options to be used for this mount. The *optlen* argument specifies the length of the buffer. On completion of the `mount()` call, the options in effect for the mounted file system are returned in this buffer. If `MS_OPTIONSTR` is not specified, then the options for this mount will not appear in the mounted file systems table.

If the caller does not have all privileges available in the current zone, the `nosuid` option is automatically set on the mount point. The `restrict` option is automatically added for `autofs` mounts.

If the caller is not in the global zone, the `nodevices` option is automatically set.

The *mflag* argument is constructed by a bitwise-inclusive-OR of flags from the following list, defined in `<sys/mount.h>`.

`MS_DATA` The *dataptr* and *datalen* arguments describe a block of file system-specific binary 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 0.

`MS_GLOBAL` Mount a file system globally if the system is configured and booted as part of a cluster (see `clinfo(1M)`).

MS_NOSUID — Prevent programs that are marked set-user-ID or set-group-ID from executing (see chmod(1)). It also causes open(2) to return ENXIO when attempting to open block or character special files.

MS_OPTIONSTR — The *optptr* and *optlen* arguments describe a character buffer at address *optptr* of size *optlen*. When calling mount(), the character buffer should contain a null-terminated string of options to be passed to the file system-specific code within the operating system. On a successful return, the file system-specific code will return the list of options recognized. Unrecognized options are ignored. The format of the string is a list of option names separated by commas. Options that have values (rather than binary options such as suid or nosuid), are separated by "=" such as dev=2c4046c. Standard option names are defined in <sys/mntent.h>. Only strings defined in the "C" locale are supported. The maximum length option string that can be passed to or returned from a mount() call is defined by the MAX_MNTOPT_STR constant. The buffer should be long enough to contain more options than were passed in, as the state of any default options that were not passed in the input option string may also be returned in the recognized options list that is returned.

MS_OVERLAY — Allow the file system to be mounted over an existing file system mounted on *dir*, making the underlying file system inaccessible. If a mount is attempted on a pre-existing mount point without setting this flag, the mount will fail.

MS_RDONLY — Mount the file system for reading only. This flag should also be specified for file systems that are incapable of writing (for example, CDROM). Without this flag, writing is permitted according to individual file accessibility.

MS_REMOUNT — Remount a read-only file system as read-write.

Return Values Upon successful completion, 0 is returned. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The mount() function will fail if:

EACCES — The permission bits of the mount point do not permit read/write access or search permission is denied on a component of the path prefix.

The calling process is not the owner of the mountpoint.

The mountpoint is not a regular file or a directory and the caller does not have all privileges available in a its zone.

The special device device does not permit read access in the case of read-only mounts or read-write access in the case of read/write mounts.

EBUSY	The *dir* argument is currently mounted on, is someone's current working directory, or is otherwise busy; or the device associated with *spec* is currently mounted.
EFAULT	The *spec*, *dir*, *fstype*, *dataptr*, or *optptr* argument points outside the allocated address space of the process.
EINVAL	The super block has an invalid magic number, the *fstype* is invalid, or *dir* is not an absolute path.
ELOOP	Too many symbolic links were encountered in translating *spec* or *dir*.
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.
ENOLINK	The *path* argument points to a remote machine and the link to that machine is no longer active.
ENOSPC	The file system state in the super-block is not FsOKAY and *mflag* requests write permission.
ENOTBLK	The *spec* argument is not a block special device.
ENOTDIR	The *dir* argument is not a directory, or a component of a path prefix is not a directory.
ENOTSUP	A global mount is attempted (the MS_GLOBAL flag is set in *mflag*) on a machine which is not booted as a cluster; a local mount is attempted and *dir* is within a globally mounted file system; or a remount was attempted on a file system that does not support remounting.
ENXIO	The device associated with *spec* does not exist.
EOVERFLOW	The length of the option string to be returned in the *optptr* argument exceeds the size of the buffer specified by *optlen*.
EPERM	The {PRIV_SYS_MOUNT} privilege is not asserted in the effective set of the calling process.
EREMOTE	The *spec* argument is remote and cannot be mounted.
EROFS	The *spec* argument is write protected and *mflag* requests write permission.

Usage The mount() function can be invoked only by processes with appropriate privileges.

See Also mount(1M), umount(2), mnttab(4)

Notes MS_OPTIONSTR-type option strings should be used.

Some flag bits set file system options that can also be passed in an option string. Options are first set from the option string with the last setting of an option in the string determining the value to be set by the option string. Any options controlled by flags are then applied, overriding any value set by the option string.

Name mprotect – set protection of memory mapping

Synopsis

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

int mprotect(void *addr, size_t len, int prot);
```

Description The `mprotect()` function changes the access protections on the mappings specified by the range [*addr, addr* + *len*), rounding *len* up to the next multiple of the page size as returned by `sysconf(3C)`, to be that specified by *prot*. Legitimate values for *prot* are the same as those permitted for `mmap(2)` 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 */
```

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.

Return Values Upon successful completion, `mprotect()` returns `0`. Otherwise, it returns −`1` and sets `errno` to indicate the error.

Errors The `mprotect()` function will fail if:

`EACCES` The *prot* argument specifies a protection that violates the access permission the process has to the underlying memory object.

`EINVAL` The *len* argument has a value equal to 0, or *addr* is not a multiple of the page size as returned by `sysconf(3C)`.

`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.

The `mprotect()` function may fail if:

`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 in the now-writable address range.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also mmap(2), plock(3C), mlock(3C), mlockall(3C), sysconf(3C), attributes(5), standards(5)

Name msgctl – message control operations

Synopsis

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

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

Description The `msgctl()` function 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 /* access permission bits only */
msg_qbytes
```

This command can be executed only by a process that has either the {`PRIV_IPC_OWNER`} privilege or an effective user ID equal to the value of `msg_perm.cuid` or `msg_perm.uid` in the data structure associated with *msqid*. Only a process with the {`PRIV_SYS_IPC_CONFIG`} privilege 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 either with appropriate privileges asserted in the effective set or equal to the value of `msg_perm.cuid` or `msg_perm.uid` in the data structure associated with *msqid*. The *buf* argument is ignored.

Return Values Upon successful completion, `msgctl()` returns `0`. Otherwise, it returns `−1` and sets `errno` to indicate the error.

Errors The `msgctl()` function will fail if:

`EACCES` The *cmd* argument is `IPC_STAT` and operation permission is denied to the calling process (see `Intro(2)`).

`EFAULT` The *buf* argument points to an illegal address.

`EINVAL` The *msqid* argument is not a valid message queue identifier; or the *cmd* argument is not a valid command or is `IPC_SET` and `msg_perm.uid` or `msg_perm.gid` is not valid.

`EOVERFLOW` The *cmd* argument is `IPC_STAT` and *uid* or *gid* is too large to be stored in the structure pointed to by *buf*.

EPERM — The *cmd* argument is IPC_RMID or IPC_SET, the {PRIV_SYS_IPC_OWNER} privilege is not asserted in the effective set of the calling process, and is not equal to the value of msg_perm.cuid or msg_perm.uid in the data structure associated with *msqid*.

The *cmd* argument is IPC_SET, an attempt is being made to increase to the value of msg_qbytes, and the {PRIV_SYS_IPC_CONFIG} privilege is not asserted in the effective set of the calling process.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also Intro(2), msgget(2), msgrcv(2), msgsnd(2), attributes(5), privileges(5), standards(5)

Name msgget – get message queue

Synopsis

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

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

Description The `msgget()` argument 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. See NOTES.

Return Values Upon successful completion, a non-negative integer representing a message queue identifier is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `msgget()` function will fail if:

`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 NOTES.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also rctladm(1M), Intro(2), msgctl(2), msgrcv(2), msgsnd(2), setrctl(2), ftok(3C), attributes(5), standards(5)

Notes The system-defined limit used to initialize `msg_qbytes` is the minimum enforced value of the calling process's `process.max-msg-qbytes` resource control.

The system-imposed limit on the number of message queue identifiers is maintained on a per-project basis using the `project.max-msg-ids` resource control. The `zone.max-msg-ids` resource control restricts the total amount of message queue identifiers that can be allocated by a zone.

See rctladm(1M) and setrctl(2) for information about using resource controls.

Name msgids – discover all message queue identifiers

Synopsis

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

int msgids(int *buf, uint_t nids, uint_t  *pnids);
```

Description The `msgids()` function copies all active message queue identifiers from the system into the user-defined buffer specified by *buf*, provided that the number of such identifiers is not greater than the number of integers the buffer can contain, as specified by *nids*. If the size of the buffer is insufficient to contain all of the active message queue identifiers in the system, none are copied.

Whether or not the size of the buffer is sufficient to contain all of them, the number of active message queue identifiers in the system is copied into the unsigned integer pointed to by *pnids*.

If *nids* is 0 or less than the number of active message queue identifiers in the system, *buf* is ignored.

Return Values Upon successful completion, `msgids()` returns `0`. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `msgids()` function will fail if:

`EFAULT` The *buf* or *pnids* argument points to an illegal address.

Usage The `msgids()` function returns a snapshot of all the active message queue identifiers in the system. More may be added and some may be removed before they can be used by the caller.

Examples **EXAMPLE 1** `msgids()` example

This is sample C code indicating how to use the `msgids()` function (see `msgsnap(2)`):

```
void
examine_queues()
{
     int *ids = NULL;
     uint_t nids = 0;
     uint_t n;
     int i;

     for (;;) {
          if (msgids(ids, nids, &n) != 0) {
               perror("msgids");
               exit(1);
          }
          if (n <= nids)     /* we got them all */
               break;
          /* we need a bigger buffer */
```

EXAMPLE 1 `msgids()` example *(Continued)*

```
            ids = realloc(ids, (nids = n) * sizeof (int));
        }

        for (i = 0; i < n; i++)
            process_msgid(ids[i]);

        free(ids);
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

See Also

`ipcrm(1)`, `ipcs(1)`, `Intro(2)`, `msgctl(2)`, `msgget(2)`, `msgsnap(2)`, `msgrcv(2)`, `msgsnd(2)`, `attributes(5)`

Name msgrcv – message receive operation

Synopsis

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

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz,
     long int msgtyp, int msgflg);
```

Description The `msgrcv()` function reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the user-defined buffer pointed to by *msgp*.

The *msgp* argument points to a user-defined buffer that must contain first a field of type `long int` that will specify the type of the message, and then a data portion that will hold the data bytes of the message. The structure below is an example of what this user-defined buffer might look like:

```
struct mymsg {
        long int    mtype;     /* message type */
        char        mtext[1];  /* message text */
}
```

The `mtype` member is the received message's type as specified by the sending process.

The `mtext` member is the text of the message.

The *msgsz* argument 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 non-zero. The truncated part of the message is lost and no indication of the truncation is given to the calling process.

The *msgtyp* argument 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.

The *msgflg* argument specifies which of the following actions is to be taken if a message of the desired type is not on the queue:

- If (*msgflg*&`IPC_NOWAIT`) is non-zero, the calling process will return immediately with a return value of −1 and `errno` set to `ENOMSG`.
- If (*msgflg*&`IPC_NOWAIT`) is 0, the calling process will suspend execution until one of the following occurs:
  - A message of the desired type is placed on the queue.
  - The message queue identifier *msqid* is removed from the system (see `msgctl(2)`); when this occurs, `errno` is set equal to `EIDRM` and −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 `sigaction(2)`.

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

- `msg_qnum` is decremented by 1.
- `msg_lrpid` is set equal to the process ID of the calling process.
- `msg_rtime` is set equal to the current time.

Return Values Upon successful completion, `msgrcv()` returns a value equal to the number of bytes actually placed into the buffer *mtext*. Otherwise, `−1` is returned, no message is received, and `errno` is set to indicate the error.

Errors The `msgrcv()` function will fail if:

`E2BIG` The value of `mtext` is greater than *msgsz* and (*msgflg*`&MSG_NOERROR`) is 0.

`EACCES` Operation permission is denied to the calling process. See Intro(2).

`EIDRM` The message queue identifier *msqid* is removed from the system.

`EINTR` The `msgrcv()` function was interrupted by a signal.

`EINVAL` The *msqid* argument is not a valid message queue identifier.

`ENOMSG` The queue does not contain a message of the desired type and (*msgflg*`&IPC_NOWAIT`) is non-zero.

The `msgrcv()` function may fail if:

`EFAULT` The *msgp* argument points to an illegal address.

Usage The value passed as the *msgp* argument should be converted to type `void *`.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also Intro(2), msgctl(2), msgget(2), msgsnd(2), sigaction(2), attributes(5), standards(5)

Name msgsnap – message queue snapshot operation

Synopsis

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

msgsnap(int msqid, void *buf, size_t bufsz, long msgtyp);
```

Description The `msgsnap()` function reads all of the messages of type *msgtyp* from the queue associated with the message queue identifier specified by *msqid* and places them in the user-defined buffer pointed to by *buf*.

The *buf* argument points to a user-defined buffer that on return will contain first a buffer header structure:

```
struct msgsnap_head {
     size_t  msgsnap_size;   /* bytes used/required in the buffer */
     size_t  msgsnap_nmsg;   /* number of messages in the buffer */
};
```

followed by `msgsnap_nmsg` messages, each of which starts with a message header:

```
struct msgsnap_mhead {
     size_t  msgsnap_mlen;   /* number of bytes in the message */
     long    msgsnap_mtype;  /* message type */
};
```

and followed by `msgsnap_mlen` bytes containing the message contents.

Each subsequent message header is located at the first byte following the previous message contents, rounded up to a `sizeof(size_t)` boundary.

The *bufsz* argument specifies the size of *buf* in bytes. If *bufsz* is less than `sizeof(msgsnap_head)`, `msgsnap()` fails with `EINVAL`. If *bufsz* is insufficient to contain all of the requested messages, `msgsnap()` succeeds but returns with `msgsnap_nmsg` set to 0 and with `msgsnap_size` set to the required size of the buffer in bytes.

The *msgtyp* argument specifies the types of messages requested as follows:

- If *msgtyp* is 0, all of the messages on the queue are read.
- If *msgtyp* is greater than 0, all messages of type *msgtyp* are read.
- If *msgtyp* is less than 0, all messages with type less than or equal to the absolute value of *msgtyp* are read.

The `msgsnap()` function is a non-destructive operation. Upon completion, no changes are made to the data structures associated with *msqid*.

Return Values Upon successful completion, `msgsnap()` returns `0`. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `msgsnap()` function will fail if:

`EACCES` Operation permission is denied to the calling process. See `Intro(2)`.

`EINVAL` The *msqid* argument is not a valid message queue identifier or the value of *bufsz* is less than `sizeof(struct msgsnap_head)`.

`EFAULT` The *buf* argument points to an illegal address.

Usage The `msgsnap()` function returns a snapshot of messages on a message queue at one point in time. The queue contents can change immediately following return from `msgsnap()`.

Examples **EXAMPLE 1** `msgsnap()` example

This is sample C code indicating how to use the msgsnap function (see `msgids(2)`).

```
void
process_msgid(int msqid)
{
     size_t bufsize;
     struct msgsnap_head *buf;
     struct msgsnap_mhead *mhead;
     int i;

     /* allocate a minimum-size buffer */
     buf = malloc(bufsize = sizeof(struct msgsnap_head));

     /* read all of the messages from the queue */
     for (;;) {
          if (msgsnap(msqid, buf, bufsize, 0) != 0) {
               perror("msgsnap");
                    free(buf);
                    return;
          }
          if (bufsize >= buf->msgsnap_size)  /* we got them all */
               break;
          /* we need a bigger buffer */
          buf = realloc(buf, bufsize = buf->msgsnap_size);
     }

     /* process each message in the queue (there may be none) */
     mhead = (struct msgsnap_mhead *)(buf + 1);  /* first message */
     for (i = 0; i < buf->msgsnap_nmsg; i++) {
          size_t mlen = mhead->msgsnap_mlen;

          /* process the message contents */
          process_message(mhead->msgsnap_mtype, (char *)(mhead+1), mlen);

          /* advance to the next message header */
```

EXAMPLE 1 `msgsnap()` example *(Continued)*

```
            mhead = (struct msgsnap_mhead *)
                 ((char *)mhead + sizeof(struct msgsnap_mhead) +
                 ((mlen + sizeof(size_t) - 1) & ~(sizeof(size_t) - 1)));
        }

        free(buf);
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

See Also

`ipcrm(1)`, `ipcs(1)`, `Intro(2)`, `msgctl(2)`, `msgget(2)`, `msgids(2)`, `msgrcv(2)`, `msgsnd(2)`, `attributes(5)`

Name msgsnd – message send operation

Synopsis

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

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

Description The `msgsnd()` function is used to send a message to the queue associated with the message queue identifier specified by *msqid*.

The *msgp* argument points to a user-defined buffer that must contain first a field of type `long int` that will specify the type of the message, and then a data portion that will hold the data bytes of the message. The structure below is an example of what this user-defined buffer might look like:

```
struct  mymsg {
        long  mtype;     /* message type */
        char  mtext[1];  /* message text */
}
```

The `mtype` member is a non-zero positive type `long int` that can be used by the receiving process for message selection.

The `mtext` member is any text of length *msgsz* bytes. The *msgsz* argument can range from 0 to a system-imposed maximum.

The *msgflg* argument 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 the queue would exceed the maximum allowed by the system. See NOTES.

These actions are as follows:

- If (*msgflg*`&IPC_NOWAIT`) is non-zero, the message will not be sent and the calling process will return immediately.
- If (*msgflg*`&IPC_NOWAIT`) is 0, the calling process will suspend execution until one of the following occurs:
  - The condition responsible for the suspension no longer exists, in which case the message is sent.
  - The message queue identifier *msqid* is removed from the system (see `msgctl(2)`); when this occurs, `errno` is set equal to `EIDRM` and `−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 `sigaction(2)`.

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

- `msg_qnum` is incremented by 1.
- `msg_lspid` is set equal to the process ID of the calling process.
- `msg_stime` is set equal to the current time.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned, no message is sent, and `errno` is set to indicate the error.

Errors The `msgsnd()` function will fail if:

`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 non-zero.

`EIDRM` The message queue identifier *msgid* is removed from the system.

`EINTR` The `msgsnd()` function was interrupted by a signal.

`EINVAL` The value of *msqid* is not a valid message queue identifier, or the value of `mtype` is less than 1.

The value of *msgsz* is less than 0 or greater than the system-imposed limit.

The `msgsnd()` function may fail if:

`EFAULT` The *msgp* argument points to an illegal address.

Usage The value passed as the *msgp* argument should be converted to type `void *`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also `rctladm(1M)`, `Intro(2)`, `msgctl(2)`, `msgget(2)`, `msgrcv(2)`, `setrctl(2)`, `sigaction(2)`, `attributes(5)`, `standards(5)`

Notes The maximum number of messages allowed on a message queue is the minimum enforced value of the `process.max-msg-messages` resource control of the creating process at the time `msgget(2)` was used to allocate the queue.

See `rctladm(1M)` and `setrctl(2)` for information about using resource controls.

Name munmap – unmap pages of memory

Synopsis

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

int munmap(void *addr, size_t len);
```

Description The `munmap()` function removes the mappings for pages in the range [*addr, addr* + *len*), rounding the *len* argument up to the next multiple of the page size as returned by `sysconf(3C)`. If *addr* is not the address of a mapping established by a prior call to `mmap(2)`, the behavior is undefined. After a successful call to `munmap()` and before any subsequent mapping of the unmapped pages, further references to these pages will result in the delivery of a `SIGBUS` or `SIGSEGV` signal to the process.

The `mmap(2)` function often performs an implicit `munmap()`.

Return Values Upon successful completion, `munmap()` returns `0`; otherwise, it returns `−1` and sets `errno` to indicate an error.

Errors The `munmap()` function will fail if:

`EINVAL` The *addr* argument is not a multiple of the page size as returned by `sysconf(3C)`; addresses in the range [*addr, addr* + *len*) are outside the valid range for the address space of a process; or the *len* argument has a value less than or equal to `0`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also `mmap(2)`, `sysconf(3C)`, `attributes(5)`, `standards(5)`

Name nice – change priority of a process

Synopsis

```
#include <unistd.h>

int nice(int incr);
```

Description The `nice()` function allows a process to change its priority. The invoking process must be in a scheduling class that supports the `nice()`.

The `nice()` function 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 greater positive value results in lower CPU priority.

A maximum nice value of (2 * `NZERO`) −1 and a minimum nice value of 0 are imposed by the system. `NZERO` is defined in <`limits.h`> with a default value of 20. Requests for values above or below these limits result in the nice value being set to the corresponding limit. A nice value of 40 is treated as 39.

Calling the `nice()` function has no effect on the priority of processes or threads with policy `SCHED_FIFO` or `SCHED_RR`.

Only a process with the {`PRIV_PROC_PRIOCNTL`} privilege can lower the nice value.

Return Values Upon successful completion, `nice()` returns the new nice value minus `NZERO`. Otherwise, −1 is returned, the process's *nice* value is not changed, and `errno` is set to indicate the error.

Errors The `nice()` function will fail if:

`EINVAL` The `nice()` function is called by a process in a scheduling class other than time-sharing or fixed-priority.

`EPERM` The *incr* argument is negative or greater than 40 and the {`PRIV_PROC_PRIOCNTL`} privilege is not asserted in the effective set of the calling process.

Usage The `priocntl(2)` function is a more general interface to scheduler functions.

Since −1 is a permissible return value in a successful situation, an application wishing to check for error situations should set `errno` to 0, then call `nice()`, and if it returns −1, check to see if `errno` is non-zero.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also nice(1), exec(2), priocntl(2), getpriority(3C), attributes(5), privileges(5), standards(5)

Name ntp_adjtime – adjust local clock parameters

Synopsis

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

int ntp_adjtime(struct timex *tptr);
```

Description The `ntp_adjtime()` function adjusts the parameters used to discipline the local clock, according to the values in the struct `timex` pointed to by *tptr*. Before returning, it fills in the structure with the most recent values kept in the kernel.

The adjustment is effected in part by speeding up or slowing down the clock, as necessary, and in part by phase-locking onto a once-per second pulse (PPS) provided by a driver, if available.

```
struct timex {
    uint32_t modes;        /* clock mode bits (w) */
    int32_t  offset;       /* time offset (us) (rw) */
    int32_t  freq;         /* frequency offset (scaled ppm) (rw) */
    int32_t  maxerror;     /* maximum error (us) (rw) */
    int32_t  esterror;     /* estimated error (us) (rw) */
    int32_t  status;       /* clock status bits (rw) */
    int32_t  constant;     /* pll time constant (rw) */
    int32_t  precision;    /* clock precision (us) (r) */
    int32_t  tolerance;    /* clock frequency tolerance
                              (scaled ppm) (r) */
    int32_t  ppsfreq;      /* pps frequency (scaled ppm) (r) */
    int32_t  jitter;       /* pps jitter (us) (r) */
    int32_t  shift;        /* interval duration (s) (shift) (r) */
    int32_t  stabil;       /* pps stability (scaled ppm) (r) */
    int32_t  jitcnt;       /* jitter limit exceeded (r) */
    int32_t  calcnt;       /* calibration intervals (r) */
    int32_t  errcnt;       /* calibration errors (r) */
    int32_t  stbcnt;       /* stability limit exceeded (r) */
};
```

Return Values Upon successful completion, `ntp_adjtime()` returns the current clock state (see <`sys/timex.h`>). Otherwise, it returns −1 and sets `errno` to indicate the error.

Errors The `ntp_adjtime()` function will fail if:

`EFAULT` The *tptr* argument is an invalid pointer.

`EINVAL` The `constant` member of the structure pointed to by *tptr* is less than 0 or greater than 30.

`EPERM` The {`PRIV_SYS_TIME`} privilege is not asserted in the effective set of the calling process.

See Also `ntp_gettime(2)`, `privileges(5)`

See the `ntpd` man page, delivered in the `SUNWntpu` package (not a SunOS man page).

Name ntp_gettime – get local clock values

Synopsis

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

int ntp_gettime(struct ntptimeval *tptr);
```

Description The `ntp_gettime()` function reads the local clock value and dispersion, returning the information in *tptr*.

The `ntptimeval` structure contains the following members:

```
struct ntptimeval {
    struct timeval   time;       /* current time (ro) */
    int32_t          maxerror;   /* maximum error (us) (ro) */
    int32_t          esterror;   /* estimated error (us) (ro) */
};
```

Return Values Upon successful completion, `ntp_gettime()` returns the current clock state (see `<sys/timex.h>`). Otherwise, it returns −1 and sets `errno` to indicate the error.

Errors The `ntp_gettime()` function will fail if:

`EFAULT` The *tptr* argument points to an invalid address.

The `ntp_gettime()` function will fail for 32-bit interfaces if:

`EOVERFLOW` The size of the `time.tv_sec` member of the `ntptimeval` structure pointed to by *tptr* is too small to contain the correct number of seconds.

See Also `ntp_adjtime(2)`

See the `ntpd` man page, delivered in the `SUNWntpu` package (not a SunOS man page).

Name open, openat – open a file

Synopsis

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

int open(const char *path, int oflag, /* mode_t mode */);

int openat(int fildes, const char *path, int oflag,
     /* mode_t mode */);
```

Description The `open()` function establishes the connection between a file and a file descriptor. It creates an open file description that refers to a file and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to that file. The *path* argument points to a pathname naming the file.

The `openat()` function is identical to the `open()` function except that the *path* argument is interpreted relative to the starting point implied by the *fildes* argument. If the *fildes* argument has the special value `AT_FDCWD`, a relative path argument will be resolved relative to the current working directory. If the *path* argument is absolute, the *fildes* argument is ignored.

The `open()` function returns a file descriptor for the named file that is the lowest file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor does not share it with any other process in the system. The `FD_CLOEXEC` file descriptor flag associated with the new file descriptor is cleared.

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

The file status flags and file access modes of the open file description are set according to the value of *oflag*. The *mode* argument is used only when `O_CREAT` is specified (see below.)

Values for *oflag* are constructed by a bitwise-inclusive-OR of flags from the following list, defined in `<fcntl.h>`. Applications must specify exactly one of the first three values (file access modes) below in the value of *oflag*:

`O_RDONLY` Open for reading only.

`O_WRONLY` Open for writing only.

`O_RDWR` Open for reading and writing. The result is undefined if this flag is applied to a FIFO.

Any combination of the following may be used:

`O_APPEND`
If set, the file offset is set to the end of the file prior to each write.

`O_CREAT`
Create the file if it does not exist. This flag requires that the *mode* argument be specified.

If the file exists, this flag has no effect except as noted under O_EXCL below. Otherwise, the file is created with the user ID of the file set to the effective user ID of the process. The group ID of the file is set to the effective group IDs 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 (see <sys/stat.h>) of the file mode are set to the value of *mode*, modified as follows (see creat(2)): a bitwise-AND is performed on the file-mode bits and the corresponding bits in the complement of the process's file mode creation mask. Thus, all bits set in the process's file mode creation mask (see umask(2)) are correspondingly cleared in the file's permission mask. 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.h(3HEAD) definition of O_SYNC.) When bits other than the file permission bits are set, the effect is unspecified. The *mode* argument does not affect whether the file is open for reading, writing or for both.

O_DSYNC
: Write I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion.

O_EXCL
: If O_CREAT and O_EXCL are set, open() fails 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 threads executing open() naming the same filename in the same directory with O_EXCL and O_CREAT set. If O_EXCL and O_CREAT are set, and path names a symbolic link, open() fails and sets errno to EEXIST, regardless of the contents of the symbolic link. If O_EXCL is set and O_CREAT is not set, the result is undefined.

O_LARGEFILE
: If set, the offset maximum in the open file description is the largest value that can be represented correctly in an object of type off64_t.

O_NOCTTY
: If set and *path* identifies a terminal device, open() does not cause the terminal device to become the controlling terminal for the process.

O_NOFOLLOW
: If the path names a symbolic link, open() fails and sets errno to ELOOP.

O_NOLINKS
: If the link count of the named file is greater than 1, open() fails and sets errno to EMLINK.

O_NONBLOCK or O_NDELAY
: These flags can affect subsequent reads and writes (see read(2) and write(2)). If both O_NDELAY and O_NONBLOCK are set, O_NONBLOCK takes precedence.

When opening a FIFO with O_RDONLY or O_WRONLY set:

- If O_NONBLOCK or O_NDELAY is set, an open() for reading only returns without delay. An open() for writing only returns an error if no process currently has the file open for reading.
- If O_NONBLOCK and O_NDELAY are clear, an open() for reading only blocks until a thread opens the file for writing. An open() for writing only blocks the calling thread until a thread opens the file for reading.

After both ends of a FIFO have been opened, there is no guarantee that further calls to open() O_RDONLY (O_WRONLY) will synchronize with later calls to open() O_WRONLY (O_RDONLY) until both ends of the FIFO have been closed by all readers and writers. Any data written into a FIFO will be lost if both ends of the FIFO are closed before the data is read.

When opening a block special or character special file that supports non-blocking opens:

- If O_NONBLOCK or O_NDELAY is set, the open() function returns without blocking for the device to be ready or available. Subsequent behavior of the device is device-specific.
- If O_NONBLOCK and O_NDELAY are clear, the open() function blocks the calling thread until the device is ready or available before returning.

Otherwise, the behavior of O_NONBLOCK and O_NDELAY is unspecified.

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
: Write I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.

O_TRUNC
: If the file exists and is a regular file, and the file is successfully opened O_RDWR or O_WRONLY, its length is truncated to 0 and the mode and owner are unchanged. It has no effect on FIFO special files or terminal device files. Its effect on other file types is implementation-dependent. The result of using O_TRUNC with O_RDONLY is undefined.

O_XATTR
: If set in openat(), a relative path argument is interpreted as a reference to an extended attribute of the file associated with the supplied file descriptor. This flag therefore requires the presence of a legal *fildes* argument. If set in open(), the implied file descriptor is that for the current working directory. Extended attributes must be referenced with a relative path; providing an absolute path results in a normal file reference.

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.

If both the O_SYNC and O_DSYNC flags are set, the effect is as if only the O_SYNC flag was set.

If *path* refers to a STREAMS file, *oflag* may be constructed from O_NONBLOCK or O_NODELAY 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 O_NONBLOCK and O_NODELAY affect the operation of STREAMS drivers and certain functions (see read(2), getmsg(2), putmsg(2), and write(2)) applied to file descriptors associated with STREAMS files. For STREAMS drivers, the implementation of O_NONBLOCK and O_NODELAY is device-specific.

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* names the master side of a pseudo-terminal device, then it is unspecified whether open() locks the slave side so that it cannot be opened. Portable applications must call unlockpt(3C) before opening the slave side.

If the file is a regular file and the local file system is mounted with the nbmand mount option, then a mandatory share reservation is automatically obtained on the file. The share reservation is obtained as if fcntl(2) were called with *cmd* F_SHARE_NBMAND and the fshare_t values set as follows:

f_access — Set to the type of read/write access for which the file is opened.

f_deny — F_NODNY

f_id — The file descriptor value returned from open().

If *path* is a symbolic link and O_CREAT and O_EXCL are set, the link is not followed.

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

The largest value that can be represented correctly in an object of type off_t is established as the offset maximum in the open file description.

Return Values Upon successful completion, the open() function opens the file and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, −1 is returned, errno is set to indicate the error, and no files are created or modified.

Errors The `open()` and `openat()` functions will fail if:

`EACCES` Search permission is denied on a component of the path prefix.

The file exists and the permissions specified by *oflag* are denied.

The file does not exist and write permission is denied for the parent directory of the file to be created.

`O_TRUNC` is specified and write permission is denied.

The {`PRIV_FILE_DAC_SEARCH`} privilege allows processes to search directories regardless of permission bits. The {`PRIV_FILE_DAC_WRITE`} privilege allows processes to open files for writing regardless of permission bits. See `privileges(5)` for special considerations when opening files owned by UID 0 for writing. The {`PRIV_FILE_DAC_READ`} privilege allows processes to open files for reading regardless of permission bits.

`EAGAIN` A mandatory share reservation could not be obtained because the desired access conflicts with an existing `f_deny` share reservation.

`EBADF` The file descriptor provided to `openat()` is invalid.

`EDQUOT` The file does not exist, `O_CREAT` is specified, and either the directory where the new file entry is being placed cannot be extended because the user's quota of disk blocks on that file system has been exhausted, or the user's quota of inodes on the file system where the file is being created has been exhausted.

`EEXIST` The `O_CREAT` and `O_EXCL` flags are set and the named file exists.

`EILSEQ` The *path* argument includes non-UTF8 characters and the file system accepts only file names where all characters are part of the UTF-8 character codeset.

`EINTR` A signal was caught during `open()`.

`EFAULT` The *path* argument points to an illegal address.

`EINVAL` The system does not support synchronized I/O for this file, or the `O_XATTR` flag was supplied and the underlying file system does not support extended file attributes.

`EIO` The *path* argument names a STREAMS file and a hangup or error occurred during the `open()`.

`EISDIR` The named file is a directory and *oflag* includes `O_WRONLY` or `O_RDWR`.

`ELOOP` Too many symbolic links were encountered in resolving *path*.

	A loop exists in symbolic links encountered during resolution of the *path* argument.
	The `O_NOFOLLOW` flag is set and the final component of path is a symbolic link.
`EMFILE`	There are currently {`OPEN_MAX`} file descriptors open in the calling process.
`EMLINK`	The `O_NOLINKS` flag is set and the named file has a link count greater than 1.
`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 a pathname component is longer than {`NAME_MAX`}.
`ENFILE`	The maximum allowable number of files is currently open in the system.
`ENOENT`	The `O_CREAT` flag is not set and the named file does not exist; or the `O_CREAT` flag is set and either the path prefix does not exist or the *path* argument points to an empty string.
`ENOLINK`	The *path* argument points to a remote machine, and the link to that machine is no longer active.
`ENOSR`	The *path* argument names a STREAMS-based file and the system is unable to allocate a STREAM.
`ENOSPC`	The directory or file system that would contain the new file cannot be expanded, the file does not exist, and `O_CREAT` is specified.
`ENOSYS`	The device specified by *path* does not support the open operation.
`ENOTDIR`	A component of the path prefix is not a directory or a relative path was supplied to `openat()`, the `O_XATTR` flag was not supplied, and the file descriptor does not refer to a directory.
`ENXIO`	The `O_NONBLOCK` flag is set, the named file is a FIFO, the `O_WRONLY` flag is set, and no process has the file open for reading; or the named file is a character special or block special file and the device associated with this special file does not exist or has been retired by the fault management framework .
`EOPNOTSUPP`	An attempt was made to open a path that corresponds to a `AF_UNIX` socket.
`EOVERFLOW`	The named file is a regular file and either `O_LARGEFILE` is not set and the size of the file cannot be represented correctly in an object of type `off_t` or `O_LARGEFILE` is set and the size of the file cannot be represented correctly in an object of type `off64_t`.

EROFS The named file resides on a read-only file system and either O_WRONLY, O_RDWR, O_CREAT (if file does not exist), or O_TRUNC is set in the *oflag* argument.

The openat() function will fail if:

EBADF The *fildes* argument is not a valid open file descriptor or is not AT_FTCWD.

The open() function may fail if:

EAGAIN The *path* argument names the slave side of a pseudo-terminal device that is locked.

EINVAL The value of the *oflag* argument is not valid.

ENAMETOOLONG Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

ENOMEM The *path* argument names a STREAMS file and the system is unable to allocate resources.

ETXTBSY The file is a pure procedure (shared text) file that is being executed and *oflag* is O_WRONLY or O_RDWR.

Examples

EXAMPLE 1 Open a file for writing by the owner.

The following example opens the file /tmp/file, either by creating it if it does not already exist, or by truncating its length to 0 if it does exist. If the call creates a new file, the access permission bits in the file mode of the file are set to permit reading and writing by the owner, and to permit reading only by group members and others.

If the call to open() is successful, the file is opened for writing.

```
#include <fcntl.h>
...
int fd;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
char *filename = "/tmp/file";
...
fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, mode);
...
```

EXAMPLE 2 Open a file using an existence check.

The following example uses the open() function to try to create the LOCKFILE file and open it for writing. Since the open() function specifies the O_EXCL flag, the call fails if the file already exists. In that case, the application assumes that someone else is updating the password file and exits.

EXAMPLE 2 Open a file using an existence check. *(Continued)*

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#define LOCKFILE "/etc/ptmp"
...
int pfd; /* Integer for file descriptor returned by open() call. */
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
        S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
        fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
        exit(1);
}
...
```

EXAMPLE 3 Open a file for writing.

The following example opens a file for writing, creating the file if it does not already exist. If the file does exist, the system truncates the file to zero bytes.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#define LOCKFILE "/etc/ptmp"
...
int pfd;
char filename[PATH_MAX+1];
...
if ((pfd = open(filename, O_WRONLY | O_CREAT | O_TRUNC,
        S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
        perror("Cannot open output file\n"); exit(1);
}
...
```

Usage

The `open()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`. Note that using `open64()` is equivalent to using `open()` with `O_LARGEFILE` set in *oflag*.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe
Standard	For `open()`, see `standards(5)`.

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)`, `attropen(3C)`, `fcntl.h(3HEAD)`, `stat.h(3HEAD)`, `unlockpt(3C)`, `attributes(5)`, `lf64(5)`, `privileges(5)`, `standards(5)`, `connld(7M)`, `streamio(7I)`

Notes Hierarchical Storage Management (HSM) file systems can sometimes cause long delays when opening a file, since HSM files must be recalled from secondary storage.

Name pause – suspend process until signal

Synopsis

```
#include <unistd.h>

int pause(void);
```

Description The `pause()` function 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.

Return Values Since `pause()` suspends thread execution indefinitely unless interrupted by a signal, there is no successful completion return value. If interrupted, it returns `−1` and sets `errno` to indicate the error.

Errors The `pause()` function will fail if:

`EINTR` A signal is caught by the calling process and control is returned from the signal-catching function.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `alarm(2)`, `kill(2)`, `signal(3C)`, `wait(3C)`, `attributes(5)`, `standards(5)`

Name pcsample – program execution time profile

Synopsis

```
#include <pcsample.h>

long pcsample(uintptr_t samples[], long nsamples);
```

Description The `pcsample()` function provides CPU-use statistics by profiling the amount of CPU time expended by a program.

For profiling dynamically-linked programs and 64-bit programs, it is superior to the `profil(2)` function, which assumes that the entire program is contained in a small, contiguous segment of the address space, divides this segment into "bins", and on each clock tick increments the counter in the bin where the program is currently executing. With shared libraries creating discontinuous program segments spread throughout the address space, and with 64-bit address spaces so large that the size of "bins" would be measured in megabytes, the `profil()` function is of limited value.

The `pcsample()` function is passed an array *samples* containing *nsamples* pointer-sized elements. During program execution, the kernel samples the program counter of the process, storing unadulterated values in the array on each clock tick. The kernel stops writing to the array when it is full, which occurs after *nsamples* / `HZ` seconds of process virtual time. The `HZ` value is obtained by invoking the call `sysconf(_SC_CLK_TCK)`. See `sysconf(3C)`.

The sampling can be stopped by a subsequent call to `pcsample()` with the *nsamples* argument set to 0. Like `profil()`, sampling continues across a call to `fork(2)`, but is disabled by a call to one of the `exec` family of functions (see `exec(2)`). It is also disabled if an update of the *samples*`[ ]` array causes a memory fault.

Return Values The `pcsample()` function always returns `0` the first time it is called. On subsequent calls, it returns the number of samples that were stored during the previous invocation. If *nsamples* is invalid, it returns `−1` and sets `errno` to indicate the error.

Errors The `pcsample()` function will fail if:

`EINVAL` The value of *nsamples* is not valid.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe
Interface Stability	Stable

See Also `exec(2)`, `fork(2)`, `profil(2)`, `sysconf(3C)`, `attributes(5)`

Name pipe – create an interprocess channel

Synopsis

```
#include <unistd.h>

int pipe(int fildes[2]);
```

Description The `pipe()` function 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`, `O_NONBLOCK`, and `FD_CLOEXEC` flags are cleared on both file descriptors. The `fcntl(2)` function can be used to set these flags.

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.

Upon successful completion `pipe()` marks for update the `st_atime`, `st_ctime`, and `st_mtime` fields of the pipe.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `pipe()` function will fail if:

`EMFILE` More than {`OPEN_MAX`} file descriptors are already in use by this process.

`ENFILE` The number of simultaneously open files in the system would exceed a system-imposed limit.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `sh(1)`, `fcntl(2)`, `fstat(2)`, `getmsg(2)`, `poll(2)`, `putmsg(2)`, `read(2)`, `write(2)`, `attributes(5)`, `standards(5)`, `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(2)` 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 <poll.h>

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

Description The `poll()` function provides applications with a mechanism for multiplexing input/output over a set of file descriptors. For each member of the array pointed to by *fds*, `poll()` examines the given file descriptor for the event(s) specified in *events*. The number of `pollfd` structures in the *fds* array is specified by *nfds*. The `poll()` function identifies those file descriptors on which an application can read or write data, or on which certain events have occurred.

The *fds* argument 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 member for each open file descriptor of interest. The array's members are `pollfd` structures, which contain the following members:

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

The `fd` member specifies an open file descriptor and the `events` and `revents` members are bitmasks constructed by a logical OR operation 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 in `revents` even if the message is of zero length.

`POLLRDNORM` Normal data (priority band equals 0) may be read without blocking. For streams, this flag is set in `revents` 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 in `revents` even if the message is of zero length.

`POLLPRI` High priority data may be received without blocking. For streams, this flag is set in `revents` even if the message is of zero length.

`POLLOUT` Normal data (priority band equals 0) 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` member.

`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` member.

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

If the value fd is less than 0, 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 member 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 −1, poll() blocks until a requested event occurs or until the call is interrupted. The poll() function is not affected by the O_NDELAY and O_NONBLOCK flags.

The poll() function supports regular files, terminal and pseudo-terminal devices, streams-based files, FIFOs and pipes. The behavior of poll() on elements of *fds* that refer to other types of file is unspecified.

The poll() function supports sockets.

A file descriptor for a socket that is listening for connections will indicate that it is ready for reading, once connections are available. A file descriptor for a socket that is connecting asynchronously will indicate that it is ready for writing, once a connection has been established.

Regular files always poll() TRUE for reading and writing.

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 member is non-zero). A value of 0 indicates that the call timed out and no file descriptors have been selected. Upon failure, −1 is returned and errno is set to indicate the error.

Errors The poll() function will fail if:

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}, or one of the fd members refers to a stream or multiplexer that is linked (directly or indirectly) downstream from a multiplexer.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also Intro(2), getmsg(2), getrlimit(2), putmsg(2), read(2), write(2), select(3C), attributes(5), standards(5), chpoll(9E)

STREAMS Programming Guide

Notes Non-STREAMS drivers use chpoll(9E) to implement poll() on these devices.

Name p_online – return or change processor operational status

Synopsis

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

int p_online(processorid_t processorid, int flag);
```

Description The `p_online()` function changes or returns the operational status of processors. The state of the processor specified by the *processorid* argument is changed to the state represented by the *flag* argument.

Legal values for *flag* are `P_STATUS`, `P_ONLINE`, `P_OFFLINE`, `P_NOINTR`, `P_FAULTED`, `P_SPARE`, and `P_FORCED`.

When *flag* is `P_STATUS`, no processor status change occurs, but the current processor status is returned.

The `P_ONLINE`, `P_OFFLINE`, `P_NOINTR`, `P_FAULTED`, and `P_SPARE` values for *flag* refer to valid processor states. The `P_OFFLINE`, `P_SPARE`, and `P_FAULTED` processor states can be combined with the `P_FORCED` flag.

A processor in the `P_ONLINE` state is allowed to process LWPs (lightweight processes) and perform system activities. The processor is also interruptible by I/O devices attached to the system.

A processor in the `P_OFFLINE` state is not allowed to process LWPs. The processor is as inactive as possible. If the hardware supports such a feature, the processor is not interruptible by attached I/O devices.

A processor in the `P_NOINTR` state is allowed to process LWPs, but it is not interruptible by attached I/O devices. Typically, interrupts, when they occur are routed to other processors in the system. Not all systems support putting a processor into the `P_NOINTR` state. It is not permitted to put all the processors of a system into the `P_NOINTR` state. At least one processor must always be available to service system clock interrupts.

A processor in the `P_SPARE` state is not allowed to process LWPs. In many respects the `P_SPARE` state is similiar to the `P_OFFLINE` state, but describes a processor that is available for reactivation by management tools without administrator intervention.

A processor in the `P_FAULTED` state is not allowed to process LWPs. In many respects the `P_FAULTED` state is similiar to the `P_OFFLINE` state, but describes a processor that has been diagnosed as faulty. The privileged caller can change the state of the processor from `P_FAULTED` to any of the other states, but since the processor might generate additional errors, electing to reactivate such a processor should be carefully considered.

Forced processor state transition can be requested if a new processor state is specified with the bitwise-inclusive OR of the special `P_FORCED` flag. Forcing transition of a processor to the `P_OFFLINE`, `P_SPARE`, or `P_FAULTED` state revokes processor bindings for all threads that were

previously bound to that processor with processor_bind(2). There is no guarantee that a forced processor state transition always succeeds.

Processor numbers are integers, greater than or equal to 0, and are defined by the hardware platform. Processor numbers are not necessarily contiguous, but "not too sparse." Processor numbers should always be printed in decimal.

The maximum possible *processorid* value can be determined by calling sysconf(_SC_CPUID_MAX). The list of valid processor numbers can be determined by calling p_online() with *processorid* values from 0 to the maximum returned by sysconf(_SC_CPUID_MAX). The EINVAL error is returned for invalid processor numbers. See EXAMPLES below.

Return Values On successful completion, the value returned is the previous state of the processor, P_ONLINE, P_OFFLINE, P_NOINTR, P_FAULTED, P_SPARE, or P_POWEROFF. Otherwise, −1 is returned, the CPU state remains unchanged, and errno is set to indicate the error.

Errors The p_online() function will fail if:

EBUSY — The *flag* was P_OFFLINE or P_SPARE and the specified processor is the only on-line processor, there are currently LWPs bound to the processor, or the processor performs some essential function that cannot be performed by another processor.

The *flag* was P_NOINTR and the specified processor is the only interruptible processor in the system, or it handles interrupts that cannot be handled by another processor.

The specified processor is powered off and cannot be powered on because some platform- specific resource is not available.

EINVAL — A non-existent processor ID was specified or *flag* was invalid.

The caller is in a non-global zone, the pools facility is active, and the processor is not a member of the zone's pool's processor set.

ENOTSUP — The specified processor is powered off, and the platform does not support power on of individual processors.

EPERM — The flag was not P_STATUS and the {PRIV_SYS_RES_CONFIG} privilege is not asserted in the effective set of the calling process.

Examples **EXAMPLE 1** List the legal processor numbers.

The following code sample will list the legal processor numbers:

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

EXAMPLE 1 List the legal processor numbers. *(Continued)*

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

int
main()
{
        processorid_t i, cpuid_max;
        cpuid_max = sysconf(_SC_CPUID_MAX);
        for (i = 0; i <= cpuid_max; i++) {
              if (p_online(i, P_STATUS) != -1)
                        printf("processor %d present\n", i);
        }
        return (0);
}
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also

pooladm(1M), psradm(1M), psrinfo(1M), zoneadm(1M), processor_bind(2), processor_info(2), pset_create(2), sysconf(3C), attributes(5), privileges(5)

Name priocntl – process scheduler control

Synopsis

```
#include <sys/types.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>
#include <sys/iapriocntl.h>
#include <sys/fsspriocntl.h>
#include <sys/fxpriocntl.h>

long priocntl(idtype_t idtype, id_t id, int cmd, /* arg */ ...);
```

Description The `priocntl()` function 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 classes currently supported are the realtime class, the time-sharing class, the fair-share class, and the fixed-priority 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)` function and the `exec(2)` family of functions. The `priocntl()` function 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 realtime LWP runs before any other LWP. Therefore, inappropriate use of realtime LWP can have a dramatic negative impact on system performance.

The `priocntl()` function provides an interface for specifying a process, set of processes, or an LWP to which the function applies. The `priocntlset(2)` function 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_ALL` The `priocntl()` function applies to all existing LWPs. The value of *id* is ignored. The permission restrictions described below still apply.

`P_CID` The *id* argument is a class ID (returned by the `priocntl()` `PC_GETCID` command as explained below). The `priocntl()` function applies to all LWPs in the specified class.

`P_GID` The *id* argument is a group ID. The `priocntl()` function applies to all LWPs with this effective group ID.

`P_LWPID` The *id* argument is an LWP ID. The *priocntl* function applies to the LWP with the specified ID within the calling process.

P_PGID	The *id* argument is a process group ID. The priocntl() function applies to all LWPs currently associated with processes in the specified process group.
P_PID	The *id* argument is a process ID specifying a single process. The priocntl() function applies to all LWPs currently associated with the specified process.
P_PPID	The *id* argument is a parent process ID. The priocntl() function applies to all LWPs currently associated with processes with the specified parent process ID.
P_PROJID	The *id* argument is a project ID. The priocntl() function applies to all LWPs with this project ID.
P_SID	The *id* argument is a session ID. The priocntl() function applies to all LWPs currently associated with processes in the specified session.
P_TASKID	The *id* argument is a task ID. The priocntl() function applies to all LWPs currently associated with processes in the specified task.
P_UID	The *id* argument is a user ID. The priocntl() function applies to all LWPs with this effective user ID.
P_ZONEID	The *id* argument is a zone ID. The priocntl() function applies to all LWPs with this zone ID.
P_CTID	The *id* argument is a process contract ID. The priocntl() function applies to all LWPs with this process contract ID.

An *id* value of P_MYID can be used in conjunction with the *idtype* value to specify the LWP ID, parent process ID, process group ID, session ID, task ID, class ID, user ID, group ID, project ID, zone ID, or process contract ID of the calling LWP.

To change the scheduling parameters of an LWP (using the PC_SETPARMS or PC_SETXPARMS command as explained below) , the real or effective user ID of the LWP calling priocntl() must match the real or the calling LWP must have sufficient privileges. These are the minimum permission requirements enforced for all classes. An individual class might impose additional permissions requirements when setting LWPs to that class and/or when setting class-specific scheduling parameters.

Two special scheduling classes, SYS and SDC, exist 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 or SDC. In addition, any processes in the SYS of SDC classes 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 0 would specify all processes with a user ID of 0 except processes in the SYS and SDC classes and (if changing the parameters using PC_SETPARMS or PC_SETXPARMS) the init(1M) process.

The init process is a special case. 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 can be assigned to any class configured on the system, but the time-sharing class is almost always the appropriate choice. (Other choices might be highly undesirable. See the *System Administration Guide: Basic Administration* for more information.)

The data type and value of *arg* are specific to the type of command specified by *cmd*.

A `pcinfo_t` structure with the following members, defined in `<sys/priocntl.h>`, is used by the `PC_GETCID` and `PC_GETCLINFO` commands.

```
id_t   pc_cid;                  /* Class id */
char   pc_clname[PC_CLNMSZ];    /* Class name */
int    pc_clinfo[PC_CLINFOSZ];  /* Class information */
```

The *pc_cid* member is a class ID returned by the `priocntl()` `PC_GETCID` command.

The *pc_clname* member is a buffer of size `PC_CLNMSZ`, defined in `<sys/priocntl.h>`, used to hold the class name: `RT` for realtime, `TS` for time-sharing, `IA`for interactive, `FSS` for fair-share, or `FX` for fixed-priority. Each string is null-terminated.

The *pc_clinfo* member 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 (`REALTIME CLASS`, `TIME-SHARING CLASS`, `INTERACTIVE CLASS`, `FAIR-SHARE CLASS`, or `FIXED-PRIORITY CLASS`) below.

A `pcparms_t` structure with the following members, defined in `<sys/priocntl.h>`, is used by the `PC_SETPARMS` and `PC_GETPARMS` commands.

```
id_t  pc_cid;                  /* LWP class */
int   pc_clparms[PC_CLPARMSZ];  /* Class-specific params */
```

The *pc_cid* member is a class ID returned by the `priocntl()` `PC_GETCID` command. 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>`.

The `PC_SETXPARMS` and `PC_GETXPARMS` commands exploit the `varargs` declaration of `priocntl()`. The argument following the command code is a class name: `RT` for realtime, `TS` for time-sharing, `IA` for interactive, `FSS` for fair-share, or `FX` for fixed-priority. The parameters after the class name build a chain of (key, value) pairs, where the key determines the meaning of the value within the pair. When using `PC_GETXPARMS`, the value associated with the key is always a pointer to a scheduling parameter. In contrast, when using `PC_SETXPARMS` the scheduling parameter is given as a direct value. A key value of `0` terminates the sequence and all further keys or values are ignored.

The PC_SETXPARMS and PC_GETXPARMS commands are more flexible than PC_SETPARMS and PC_GETPARMS and should replace PC_SETPARMS and PC_GETPARMS on a long-term basis.

Commands Available priocntl() commands are:

PC_ADMIN
: This command provides functionality needed for the implementation of the dispadmin(1M) utility. It is not intended for general use by other applications.

PC_DONICE
: Set or get nice value of the specified LWP(s) associated with the specified process(es). When this command is used with the *idtype* of P_LWPID, it sets the nice value of the LWP. The *arg* argument points to a structure of type pcnice_t. The *pc_val* member specifies the nice value and the *pc_op* specifies the type of the operation.

  When *pc_op* is set to PC_GETNICE, priocntl() sets the *pc_val* to the highest priority (lowest numerical value) pertaining to any of the specified LWPs.

  When *pc_op* is set to PC_SETNICE, priocntl() sets the nice value of all LWPs in the specified set to the value specified in *pc_val* member of pcnice_t structure.

  The priocntl() function returns −1 with errno set to EPERM if the calling LWP doesn't have appropriate permissions to set or get nice values 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_GETCID
: Get class ID and class attributes for a specific class given the 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>, <sys/tspriocntl.h>, <sys/iapriocntl.h>, <sys/fsspriocntl.h>, or <sys/fxpriocntl.h> header 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. The *pc_cid* member 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>`, `<sys/tspriocntl.h>`, `<sys/iapriocntl.h>`, `<sys/fsspriocntl.h>`, or `<sys/fxpriocntl.h>` header 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_GETPARMS`
Get the class and/or class-specific scheduling parameters of an LWP. The *arg* member 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_GETXPARMS`
Get the class or class-specific scheduling parameters of an LWP. The class name (first argument after `PC_GETXPARMS`) specifies the class and the (key, value) pair sequence contains a pointer to the class-specific parameters. The keys and the types of the class-specific parameter data are described below and can also be found in the class-specific headers `<sys/rtpriocntl.h>`, `<sys/tspriocntl.h>`, `<sys/iapriocntl.h>`, `<sys/fsspriocntl.h>`, and `<sys/fxpriocntl.h>`. If the specified class is 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 given (key, value) pair buffers. If the LWP specified does not exist or does not belong to the specified class, `priocntl()` returns −1 and `errno` is set to `ESRCH`.

If the class name specifies a configured class and a set of LWPs is given, the scheduling parameters of one of the specified LWPs belonging to the specified class are returned 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, `priocntl()` returns −1 and `errno` is set to `ESRCH`.

If the class name is a null pointer, a single process or LWP is specified, and a (key, value) pair for a class name request is given, `priocntl()` fills the buffer pointed to by value with the class name of the specified process or LWP. The key for the class name request is `PC_KY_CLNAME` and the class name buffer should be declared as:

```
char   pc_clname[PC_CLNMSZ];     /* Class name */
```

`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. The *arg* argument points to a structure of type `pcparms_t`. The *pc_cid* member 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>`, `<sys/tspriocntl.h>`, `<sys/iapriocntl.h>`, `<sys/fsspriocntl.h>`, or `<sys/fxpriocntl.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 might or might 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. The `priocntl()` function 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_SETXPARMS`
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 `P_LWPID` as *idtype*, it will set the class and class-specific scheduling parameters of the LWP. The class name (first argument after `PC_SETXPARMS`) specifies the class to be changed and the following (key, value) pair sequence contains the class-specific parameters to be changed. Only those (key,value) pairs whose scheduling behavior is to change must be specified. The keys and the types of the class-specific parameter data are described below and can also be found in the class-specific header files `<sys/rtpriocntl.h>`, `<sys/tspriocntl.h>`, `<sys/iapriocntl.h>`, `<sys/fsspriocntl.h>`, and `<sys/fxpriocntl.h>`.

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 might or might not continue through the set of LWPs, depending on the nature of the error. If the error is related to permissions (`EPERM`), `priocntl()` continues to reset the parameters for all target LWPs where the calling LWP has appropriate permissions. The `priocntl()` function returns `−1` and `errno` is set to `EPERM` when the operation failed for one or more of the target LWPs. All errors other than `EPERM` result in an immediate termination of `priocntl()`.

Realtime Class The realtime 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 realtime 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 realtime LWP is given CPU service before any LWP belonging to any other class.

The realtime class has a range of realtime priority (*rt_pri*) values that can be assigned to an LWP within the class. Realtime 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 realtime scheduling policy is a fixed priority policy. The scheduling priority of a realtime 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 realtime 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 realtime LWP relative to other LWPs within its class. Numerically higher *rt_pri* values represent higher priorities. Since the realtime class controls the highest range of scheduling priorities in the system, it is guaranteed that the runnable realtime 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 realtime class. The time quantum value specifies the maximum amount of time an LWP can run assuming that it does not complete or enter a resource or event wait state (*sleep*). If another LWP becomes runnable at a higher priority, the currently running LWP might be preempted before receiving its full time quantum.

The realtime quantum signal can be used for the notification of runaway realtime processes about the consumption of their time quantum. Those processes, which are monitored by the realtime time quantum signal, receive the configured signal in the event of time quantum expiration. The default value (`0`) of the time quantum signal will denote no signal delivery and a positive value will denote the delivery of the signal specified by the value. The realtime quantum signal can be set with the `priocntl()` `PC_SETXPARMS` command and displayed with the `priocntl()` `PC_GETXPARMS` command as explained below.

The system's process scheduler keeps the runnable realtime LWPs on a set of scheduling queues. There is a separate queue for each configured realtime priority and all realtime 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). Realtime LWPs that wake up after sleeping, LWPs that change to the realtime class from some other class, LWPs that 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) function call by a realtime 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 rtinfo_t structure with the following members, defined in <sys/rtpriocntl.h>, defines the format used for the attribute data for the realtime class.

```
short    rt_maxpri;      /* Maximum realtime priority */
```

The priocntl() PC_GETCID and PC_GETCLINFO commands return realtime class attributes in the *pc_clinfo* buffer in this format.

The rt_maxpri member specifies the configured maximum rt_pri value for the realtime class. If *rt_maxpri* is x, the valid realtime priorities range from 0 to *x*.

A rtparms_t structure with the following members, defined in <sys/rtpriocntl.h>, defines the format used to specify the realtime class-specific scheduling parameters of an LWP.

```
short    rt_pri;       /* Real-Time priority */
uint_t   rt_tqsecs;    /* Seconds in time quantum */
int      rt_tqnsecs;   /* Additional nanoseconds in quantum */
```

When using the priocntl() PC_SETPARMS or PC_GETPARMS commands, if pc_cid specifies the realtime class, the data in the *pc_clparms* buffer are in this format.

These commands can be used to set the realtime 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. 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 realtime priority value. To change the time quantum of an LWP without setting the priority or affecting the LWP's position on the queue, the *rt_pri* member 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 realtime from some other class results in the realtime priority being set to 0.

For the priocntl() PC_GETPARMS command, if *pc_cid* specifies the realtime class and more than one realtime LWP is specified, the scheduling parameters of the realtime 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* members 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 member results in an error return with errno set to EINVAL. Although the resolution of the *tq_nsecs* member 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 INT_MAX1 ticks. The INT_MAX value is defined in <limits.h>. Requesting a quantum greater than this maximum results in an error return with errno set to ERANGE, although infinite quantums can be requested using a special value as explained below. Requesting a time quantum of 0 by setting both *rt_tqsecs* and rt_tqnsecs to 0 results in an error return with errno set to EINVAL.

The *rt_tqnsecs* member 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 not set the time quantum. This value is useful when you wish to change the realtime priority of an LWP without affecting the time quantum. Specifying this value when changing the class of an LWP to realtime from some other class is equivalent to specifying RT_TQDEF.

When using the priocntl() PC_SETXPARMS or PC_GETXPARMS commands, the first argument after the command code must be the class name of the realtime class (RT) . The next arguments are formed as (key, value) pairs, terminated by a 0 key. The definition for the keys of the realtime class can be found in <sys/rtpriocntl.h>. A repeated specification of the same key results in an error return and errno set to EINVAL.

Key	Value Type	Description
RT_KY_PRI	pri_t	realtime priority
RT_KY_TQSECS	uint_t	seconds in time quantum
RT_KY_TQNSECS	int	nanoseconds in time quantum
RT_KY_TQSIG	int	realtime time quantum signal

When using the priocntl() PC_GETXPARMS command, the value associated with the key is always a pointer to a scheduling parameter of the value type shown in the table above. In contrast, when using the priocntl() PC_SETXPARMS command, the scheduling parameter is given as a direct value.

A priocntl() PC_SETXPARMS command with the class name (RT) and without a following (key, value) pair will set or reset all realtime scheduling parameters of the target process(es) to their default values. Changing the class of an LWP to realtime from some other class causes the parameters to be set to their default values. The default realtime priority (RT_KY_PRI) is 0. A default time quantum (RT_TQDEF) is assigned to each priority class (see rt_dptbl(4)). The default realtime time quantum signal (RT_KY_TQSIG) is 0.

The value associated with RT_KY_TQSECS is the number of seconds in the time quantum. The value associated with RT_KY_TQNSECS is the number of nanoseconds in the quantum. Specifying a value of 1,000,000,000 or greater for the number of nanoseconds results in an error return and errno is set to EINVAL. The specified time quantum 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 INT_MAX ticks, defined in <limits.h>. Requesting a quantum greater than this maximum results in an error return and errno is set to ERANGE. If seconds (RT_KY_TQSECS) but no nanoseconds (RT_KY_TQNSECS) are supplied, the number of nanoseconds is set to 0. If nanoseconds (RT_KY_TQNSECS) but no seconds (RT_KY_TQSECS) are supplied, the number of seconds is set to 0. A time quantum of 0 (seconds and nanoseconds are 0) results in an error return with errno set to EINVAL. Special values for RT_KY_TQSECS are RT_TQINF and RT_TQDEF (as described above). The priocntl() command PC_SETXPARMS knows no special value RT_NOCHANGE.

To change the class of an LWP to realtime from any other class, the LWP invoking priocntl() must have sufficient privileges. To change the priority or time quantum setting of a realtime LWP, the LWP invoking priocntl() must have sufficient privileges or must itself be a realtime LWP whose real or effective user ID matches the real of effective user ID of the target LWP.

The realtime priority and time quantum are inherited across fork(2) and the exec family of functions. When using the time quantum signal with a user-defined signal handler across the exec functions, the new image must install an appropriate user-defined signal handler before the time quantum expires. Otherwise, unpredictable behavior might result.

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 can be assigned to LWPs within the class. A *ts_upri* value of 0 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, since the *ts_upri* value is just one factor used to determine the scheduling priority of a time-sharing LWP. The system can 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) specifying the maximum *ts_upri* value that can be set for a given LWP. By default, *ts_uprilim* is 0.

A tsinfo_t 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.

The ts_maxupri member 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 tsparms_t 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 can lower its own *ts_uprilim* (or that of another LWP with the same user ID). Only a time-sharing LWP with sufficient privileges can raise a *ts_uprilim*. When changing the class of an LWP to time-sharing from some other class, sufficient privileges are required to set the initial *ts_uprilim* to a value greater than 0. Attempts by an unprivileged LWP to raise a *ts_uprilim* or set an initial *ts_uprilim* greater than 0 fail with a return value of –1 and errno set to EPERM.

Any time-sharing LWP can 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* members can be set to the special value TS_NOCHANGE, defined in <sys/tspriocntl.h>, 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* that is being set.

When using the priocntl() PC_SETXPARMS or PC_GETXPARMS commands, the first argument after the command code is the class name of the time-sharing class (TS) . The next arguments are formed as (key, value) pairs, terminated by a 0 key. The definition for the keys of the time-sharing class can be found in <sys/tspriocntl.h>. A repeated specification of the same key results in an error return and errno set to EINVAL.

Key	Value Type	Description
TS_KY_UPRILIM	pri_t	user priority limit
TS_KY_UPRI	pri_t	user priority

When using the priocntl() PC_GETXPARMS command, the value associated with the key is always a pointer to a scheduling parameter of the value type in the table above. In contrast, when using the priocntl() PC_SETXPARMS command, the scheduling parameter is given as a direct value.

A priocntl() PC_SETXPARMS command with the class name (TS) and without a following (key, value) pair will set or reset all time-sharing scheduling parameters of the target process(es) to their default values. Changing the class of an LWP to time-sharing from some other class causes the parameters to be set to their default values. The default value for the user priority limit (TS_KY_UPRILIM) is 0. The default value for the user priority (TS_KY_UPRI) is equal to the user priority limit (TS_KY_UPRILIM) that is being set.

The priocntl() command PC_SETXPARMS knows no special value TS_NOCHANGE.

The time-sharing user priority and user priority limit are inherited across fork() and the exec family of functions.

Interactive Class The interactive scheduling policy is a variation on the time-sharing scheduling policy. All that can be said about the time-sharing scheduling policy is also true for the interactive scheduling policy, with one addition: An LWP in the interactive class with its *ia_mode* value set to IA_SET_INTERACTIVE has its time-sharing priority boosted by IA_BOOST (10).

An iainfo_t structure with the following members, defined in <sys/iapriocntl.h>, defines the format used for the attribute data for the interactive class.

```
short    ia_maxupri;     /* Limits of user priority range */
```

The priocntl() PC_GETCID and PC_GETCLINFO commands return interactive class attributes in the *pc_clinfo* buffer in this format.

The *ia_maxupri* member specifies the configured maximum user priority value for the interactive class. If *ia_maxupri* is x, the valid range for both user priorities and user priority limits is from -x to +x.

A `iaparms_t` structure with the following members, defined in `<sys/iapriocntl.h>`, defines the format used to specify the interactive class-specific scheduling parameters of an LWP.

```
short    ia_uprilim;     /* Interactive user priority limit */
short    ia_upri;        /* Interactive user priority */
int      ia_mode;        /* interactive on/off */
```

When using the `priocntl()` `PC_SETPARMS` or `PC_GETPARMS` commands, if *pc_cid* specifies the interactive class, the data in the *pc_clparms* buffer is in this format.

For the `priocntl()` `PC_GETPARMS` command, if *pc_cid* specifies the interactive class and more than one interactive LWP is specified, the scheduling parameters of the interactive LWP with the highest *ia_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.

All that is said above in the TIME-SHARING CLASS section concerning manipulation of *ts_uprilim* and *ts_upri* applies equally to manipulations of *ia_uprilim* and *ia_upri* in the interactive class.

When using the `PC_SETPARMS` command, the *ia_mode* member must be set to one of the values `IA_SET_INTERACTIVE`, `IA_INTERACTIVE_OFF`, or `IA_NOCHANGE`, defined in `<sys/iapriocntl.h>`, to set the interactive mode on or off or to make no change to the interactive mode.

When using the `priocntl()` `PC_SETXPARMS` or `PC_GETXPARMS` commands, the first argument after the command code is the class name of the interactive class (IA) . The next arguments are formed as (key, value) pairs, terminated by a 0 key. The definition for the keys of the interactive class can be found in `<sys/iapriocntl.h>`. A repeated specification of the same key results in an error return and `errno` set to `EINVAL`.

Key	Value Type	Description
`IA_KY_UPRILIM`	`pri_t`	user priority limit
`IA_KY_UPRI`	`pri_t`	user priority
`IA_KY_MODE`	`int`	interactive mode

When using the `priocntl()` `PC_GETXPARMS` command, the value associated with the key is always a pointer to a scheduling parameter of the value type in the table above. In contrast, when using the `priocntl()` `PC_SETXPARMS` command, the scheduling parameter is given as a direct value.

A `priocntl()` `PC_SETXPARMS` command with the class name (IA) and without a following (key, value) pair will set or reset all interactive scheduling parameters of the target process(es) to their default values. Changing the class of an LWP to interactive from some other class causes the parameters to be set to their default values. The default value for the user priority limit (`IA_KY_UPRILIM`) is 0. The default value for the user priority (`IA_KY_UPRI`) is equal to the user priority limit (`IA_KY_UPRILIM`) that is being set. The default value for the interactive mode (`IA_KY_MODE`) is `IA_SET_INTERACTIVE`.

The `priocntl()` command `PC_SETXPARMS` knows no special value `IA_NOCHANGE`.

The interactive user priority and user priority limit are inherited across fork and the exec family of functions.

Fair-SHARE Class The fair-share scheduling policy provides a fair allocation of CPU resources among projects, independent of the number of processes they contain. Projects are given "shares" to control their quota of CPU resources. See FSS(7) for more information about how to configure shares.

The fair share class supports the notion of per-LWP user priority (see `fss_upri` below) values for compatibility with the time-sharing scheduling class. An *fss_upri* value of 0 is defined as the default base priority for the fair-share 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 fair-share class. Raising the `fss_upri` value of an LWP in the fair-share class tells the scheduler to give this LWP more CPU time slices, while lowering the `fss_upri` value tells the scheduler to give it less CPU slices. It is not guaranteed, however, that an LWP with a higher `fss_upri` value will run before one with a lower `fss_upri` value. This is because the `fss_upri` value is just one factor used to determine the scheduling priority of a fair-share LWP. The system can dynamically adjust the internal scheduling priority of a fair-share LWP based on other factors such as recent CPU usage. The fair-share scheduler attempts to provide an evenly graded effect across the whole range of user priority values.

User priority values do not interfere with project shares. That is, changing a user priority value of a process does not have any effect on its project CPU entitlement, which is based on the number of shares it is allocated in comparison with other projects.

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 *fss_uprilim* below) that specifies the maximum `fss_upri` value that can be set for a given LWP. By default, *fss_uprilim* is 0.

A `fssinfo_t` structure with the following members, defined in `<sys/fsspriocntl.h>`, defines the format used for the attribute data for the fair-share class.

```
short    fss_maxupri;    /* Limits of user priority range */
```

The priocntl() PC_GETCID and PC_GETCLINFO commands return fair-share class attributes in the *pc_clinfo* buffer in this format.

fss_maxupri specifies the configured maximum user priority value for the fair-share class. If fss_maxupri is *x*, the valid range for both user priorities and user priority limits is from -*x* to +*x*.

A fssparms_t structure with the following members, defined in <sys/fsspriocntl.h>, defines the format used to specify the fair-share class-specific scheduling parameters of an LWP.

```
short    fss_uprilim;   /* Fair-share user priority limit */
short    fss_upri;      /* Fair-share user priority */
```

When using the priocntl() PC_SETPARMS or PC_GETPARMS commands, if *pc_cid* specifies the fair-share class, the data in the *pc_clparms* buffer is in this format.

For the priocntl() PC_GETPARMS command, if *pc_cid* specifies the fair-share class and more than one fair-share LWP is specified, the scheduling parameters of the fair-share LWP with the highest *fss_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 fair-share LWP can lower its own *fss_uprilim* (or that of another LWP with the same user ID). Only a fair-share LWP with sufficient privileges can raise an *fss_uprilim*. When changing the class of an LWP to fair-share from some other class, sufficient privileges are required to enter the FSS class or to set the initial *fss_uprilim* to a value greater than 0. Attempts by an unprivileged LWP to raise an *fss_uprilim* or set an initial *fss_uprilim* greater than 0 fail with a return value of -1 and errno set to EPERM.

Any fair-share LWP can set its own *fss_upri* (or that of another LWP with the same user ID) to any value less than or equal to the LWP's *fss_uprilim*. Attempts to set the fss_upri above the fss_uprilim (and/or set the *fss_uprilim* below the *fss_upri*) result in the *fss_upri* being set equal to the *fss_uprilim*.

Either of the *fss_uprilim* or fss_upri members can be set to the special value FSS_NOCHANGE (defined in <sys/fsspriocntl.h>) to set one of the values without affecting the other. Specifying FSS_NOCHANGE for the *fss_upri* when the *fss_uprilim* is being set to a value below the current *fss_upri* causes the *fss_upri* to be set equal to the *fss_uprilim* being set. Specifying FSS_NOCHANGE for a parameter when changing the class of an LWP to fair-share (from some other class) causes the parameter to be set to a default value. The default value for the fss_uprilim is 0 and the default for the fss_upri is to set it equal to the *fss_uprilim* which is being set.

The fair-share user priority and user priority limit are inherited across fork() and the exec family of functions.

Fixed-PRIORITY Class

The fixed-priority class provides a fixed-priority preemptive scheduling policy for those LWPs requiring that the scheduling priorities do not get dynamically adjusted by the system and that the user/application have control of the scheduling priorities.

The fixed-priority class has a range of fixed-priority user priority (see `fx_upri` below) values that can be assigned to LWPs within the class. A *fx_upri* value of 0 is defined as the default base priority for the fixed-priority class. User 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 purpose of the user priority is to provide user/application control over the scheduling of processes in the fixed-priority class. For processes in the fixed-priority class, the *fx_upri* value is, for all practical purposes, equivalent to the scheduling priority of the process. The *fx_upri* value completely determines the scheduling priority of a fixed-priority process relative to other processes within its class. Numerically higher *fx_upri* values represent higher priorities.

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 *fx_uprilim* below) that specifies the maximum *fx_upri* value that can be set for a given LWP. By default, *fx_uprilim* is 0.

A structure with the following member (defined in `<sys/fxpriocntl.h>`) defines the format used for the attribute data for the fixed-priority class.

```
pri_t   fx_maxupri;      /* Maximum user priority */
```

The `priocntl()` `PC_GETCID` and `PC_GETCLINFO` commands return fixed-priority class attributes in the *pc_clinfo* buffer in this format.

The *fx_maxupri* member specifies the configured maximum user priority value for the fixed-priority class. If *fx_maxupri* is *x*, the valid range for both user priorities and user priority limits is from 0 to *x*.

A structure with the following members (defined in `<sys/fxpriocntl.h>`) defines the format used to specify the fixed-priority class-specific scheduling parameters of an LWP.

```
pri_t    fx_upri;     /* Fixed-priority user priority */
pri_t    fx_uprilim;  /* Fixed-priority user priority limit */
uint_t   fx_tqsecs;   /* seconds in time quantum */
int      fx_tqnsecs;  /* additional nanosecs in time quant */
```

When using the `priocntl()` `PC_SETPARMS` or `PC_GETPARMS` commands, if *pc_cid* specifies the fixed-priority class, the data in the *pc_clparms* buffer is in this format.

For the `priocntl()` `PC_GETPARMS` command, if *pc_cid* specifies the fixed-priority class and more than one fixed-priority LWP is specified, the scheduling parameters of the fixed-priority LWP with the highest *fx_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 fixed-priority LWP can lower its own *fx_uprilim* (or that of another LWP with the same user ID). Only a fixed-priority LWP with sufficient privileges can raise a *fx_uprilim*. When changing the class of an LWP to fixed-priority from some other class, sufficient privileges are required to set the initial *fx_uprilim* to a value greater than 0. Attempts by an unprivileged LWP to raise a *fx_uprilim* or set an initial `fx_uprilim` greater than 0 fail with a return value of -1 and errno set to `EPERM`.

Any fixed-priority LWP can set its own *fx_upri* (or that of another LWP with the same user ID) to any value less than or equal to the LWP's *fx_uprilim*. Attempts to set the *fx_upri* above the `fx_uprilim` (and/or set the *fx_uprilim* below the *fx_upri*) result in the `fx_upri` being set equal to the *fx_uprilim*.

Either of the *fx_uprilim* or *fx_upri* members can be set to the special value `FX_NOCHANGE` (defined in `<sys/fxpriocntl.h>`) to set one of the values without affecting the other. Specifying `FX_NOCHANGE` for the *fx_upri* when the *fx_uprilim* is being set to a value below the current *fx_upri* causes the *fx_upri* to be set equal to the *fx_uprilim* being set. Specifying `FX_NOCHANGE` for a parameter when changing the class of an LWP to fixed-priority (from some other class) causes the parameter to be set to a default value. The default value for the *fx_uprilim* is 0 and the default for the *fx_upri* is to set it equal to the *fx_uprilim* that is being set. The default for time quantum is dependent on the *fx_upri* and on the system configuration; see `fx_dptbl(4)`.

The *fx_tqsecs* and *fx_tqnsecs* members are used for getting or setting the time quantum associated with an LWP or group of LWPs. *fx_tqsecs* is the number of seconds in the time quantum and *fx_tqnsecs* is the number of additional nanoseconds in the quantum. For example, setting *fx_tqsecs* to 2 and *fx_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 *fx_tqnsecs* member results in an error return with `errno` set to `EINVAL`. Although the resolution of the *tq_nsecs* member 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 `INT_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 can be requested using a special value as explained below. Requesting a time quantum of 0 (setting both *fx_tqsecs* and *fx_tqnsecs* to 0) results in an error return with `errno` set to `EINVAL`.

The *fx_tqnsecs* member can also be set to one of the following special values (defined in `<sys/fxpriocntl.h>`), in which case the value of *fx_tqsecs* is ignored:

`FX_TQINF` Set an infinite time quantum.

`FX_TQDEF` Set the time quantum to the default for this priority (see `fx_dptbl(4)`).

`FX_NOCHANGE` Do not set the time quantum. This value is useful in changing the user priority of an LWP without affecting the time quantum. Specifying this value when changing the class of an LWP to fixed-priority from some other class is equivalent to specifying `FX_TQDEF`.

When using the priocntl() PC_SETXPARMS or PC_GETXPARMS commands, the first argument after the command code must be the class name of the fixed-priority class (FX) . The next arguments are formed as (key, value) pairs, terminated by a 0 key. The definition for the keys of the fixed-priority class can be found in <sys/fxpriocntl.h>. A repeated specification of the same key results in an error return and errno set to EINVAL.

Key	Value Type	Description
FX_KY_UPRILIM	pri_t	user priority limit
FX_KY_UPRI	pri_t	user priority
FX_KY_TQSECS	uint_t	seconds in time quantum
FX_KY_TQNSECS	int	nanoseconds in time quantum

When using the priocntl() PC_GETXPARMS command, the value associated with the key is always a pointer to a scheduling parameter of the value type shown in the table above. In contrast, when using the priocntl() PC_SETXPARMS command, the scheduling parameter is given as a direct value.

A priocntl() PC_SETXPARMS command with the class name (FX) and without a following (key, value) pair will set or reset all realtime scheduling parameters of the target process(es) to their default values. Changing the class of an LWP to fixed-priority from some other class causes the parameters to be set to their default values. The default value for the user priority limit (FX_KY_UPRILIM) is 0. The default value for the user priority (FX_KY_UPRI) is equal to the user priority limit (FX_KY_UPRILIM) that is being set. A default time quantum (FX_TQDEF) is assigned to each priority class (see fx_dptbl(4)).

The value associated with FX_KY_TQSECS is the number of seconds in the time quantum. The value associated with FX_KY_TQNSECS is the number of nanoseconds in the quantum. Specifying a value of 1,000,000,000 or greater for the number of nanoseconds results in an error return and errno is set to EINVAL. The specified time quantum 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 INT_MAX ticks, defined in <limits.h>. Requesting a quantum greater than this maximum results in an error return and errno is set to ERANGE. If seconds (FX_KY_TQSECS) but no nanoseconds (FX_KY_TQNSECS) are supplied, the number of nanoseconds is set to 0. If nanoseconds (FX_KY_TQNSECS) but no seconds (FX_KY_TQSECS) are supplied, the number of seconds is set to 0. A time quantum of 0 (seconds and nanoseconds are 0) results in an error return with errno set to EINVAL. Special values for FX_KY_TQSECS are FX_TQINF and FX_TQDEF (as described above). The priocntl() command PC_SETXPARMS knows no special value FX_NOCHANGE.

The fixed-priority user priority and user priority limit are inherited across fork(2) and the exec family of functions.

Return Values Unless otherwise noted above, `priocntl()` returns `0` on success. On failure, `priocntl()` returns `–1` and sets `errno` to indicate the error.

Errors The `priocntl()` function will fail if:

`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 {`PRIV_PROC_PRIOCNTL`} privilege is not asserted in the effective set of the calling LWP.

The calling LWP does not have sufficient privileges to affect the target LWP.

`ERANGE` The requested time quantum is out of range.

`ESRCH` None of the specified LWPs exist.

See Also `priocntl(1)`, `dispadmin(1M)`, `init(1M)`, `exec(2)`, `fork(2)`, `nice(2)`, `priocntlset(2)`, `fx_dptbl(4)`, `process(4)`, `rt_dptbl(4)`, `privileges(5)`

System Administration Guide: Basic Administration

Programming 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>
#include <sys/iapriocntl.h>
#include <sys/fsspriocntl.h>
#include <sys/fxpriocntl.h>

long priocntlset(procset_t *psp, int cmd, /* arg */ ...);
```

Description The `priocntlset()` function 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 */
```

The `p_lidtype` and `p_lid` members specify the ID type and ID of one (“left”) set of processes; the `p_ridtype` and `p_rid` members 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. The `p_op` member 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 `0` on success. Otherwise, it returns `−1` and sets `errno` to indicate the error.

Errors The `priocntlset()` function will fail if:

`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 {`PRIV_PROC_PRIOCNTL`} privilege is not asserted in the effective set of the calling LWP.

The calling LWP does not have sufficient privileges to affect the target LWP.

`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 `processor_bind()` function binds the LWP (lightweight process) or set of LWPs specified by *idtype* and *id* to the processor specified by *processorid*. If *obind* is not `NULL`, this function also sets the `processorid_t` variable pointed to by *obind* 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 affects all LWPs of the process with process ID (PID) *id*.

If *idtype* is `P_LWPID`, the binding affects the LWP of the current process with LWP ID *id*.

If *idtype* is `P_TASKID`, the binding affects all LWPs of all processes with task ID *id*.

If *idtype* is `P_PROJID`, the binding affects all LWPs of all processes with project ID *id*.

If *idtype* is `P_CTID`, the binding affects all LWPs of all processes with process contract ID *id*.

If *idtype* is `P_ZONEID`, the binding affects all LWPs of all processes with zone ID *id*.

If *id* is `P_MYID`, the specified LWP, process, task, or project 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 {`PRIV_PROC_OWNER`} privilege must be asserted in the effective set of the calling process or the real or effective user ID of the calling process 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.

Processor bindings are inherited across `fork(2)` and `exec(2)`.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `processor_bind()` function will fail if:

`EFAULT` The location pointed to by *obind* was not `NULL` and not writable by the user.

`EINVAL` The specified processor is not on-line, or the *idtype* argument was not `P_PID`, `P_LWPID`, `P_PROJID`, `P_TASKID`, `P_CTID`, or `P_ZONEID`.

The caller is in a non-global zone, the pools facility is active, and the processor is not a member of the zone's pool's processor set.

`ENOTSUP` Binding a system process to a processor set is not supported.

`EPERM` The {`PRIV_PROC_OWNER`} privilege is not asserted in the effective set of the calling process and its real or effective user ID does not match the real or effective user ID of one of the LWPs being bound.

`ESRCH` No processes, LWPs, or tasks were found to match the criteria specified by *idtype* and *id*.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe

See Also

`pooladm(1M)`, `psradm(1M)`, `psrinfo(1M)`, `zoneadm(1M)`, `exec(2)`, `fork(2)`, `p_online(2)`, `pset_bind(2)`, `sysconf(3C)`, `process(4)`, `project(4)`, `attributes(5)`, `privileges(5)`

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 `processor_info()` function returns the status of the processor specified by *processorid* in the `processor_info_t` structure pointed to by *infop*.

The structure `processor_info_t` contains the following members:

```
int      pi_state;
char     pi_processor_type[PI_TYPELEN];
char     pi_fputypes[PI_FPUTYPE];
int      pi_clock;
```

The `pi_state` member is the current state of the processor, either `P_ONLINE`, `P_OFFLINE`, `P_NOINTR`, `P_FAULTED`, `P_SPARE`, or `P_POWEROFF`.

The `pi_processor_type` member is a null-terminated ASCII string specifying the type of the processor.

The `pi_fputypes` member 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.

The `pi_clock` member is the processor clock frequency rounded to the nearest megahertz. It may be `0` if not known.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `processor_info()` function will fail if:

`EINVAL` An non-existent processor ID was specified.

The caller is in a non-global zone, the pools facility is active, and the processor is not a member of the zone's pool's processor set.

`EFAULT` The `processor_info_t` structure pointed to by *infop* was not writable by the user.

See Also `pooladm(1M)`, `psradm(1M)`, `psrinfo(1M)`, `zoneadm(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 The `profil()` function provides CPU-use statistics by profiling the amount of CPU time expended by a program. The `profil()` function 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.

The *buff* argument is a buffer of *bufsiz* bytes in which the histogram counts are stored in an array of `unsigned short int`. Once one of the counts reaches 32767 (the size of a `short int`), profiling stops and no more data is collected.

The *offset*, *scale*, and *bufsiz* arguments specify the region to be profiled.

The *offset* argument is effectively the start address of the region to be profiled.

The *scale* argument 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.

The values of *scale* are as follows:

- the maximum 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.

The value of *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.

The value of *bufsiz* can be computed as (*size_of_region_to_be_profiled* * *RATIO*).

Profiling is turned off by giving a *scale* value of 0 or 1, and is rendered ineffective by giving a *bufsiz* value of 0. Profiling is turned off when one of the `exec` family of functions (see `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.

Usage The `pcsample(2)` function should be used when profiling dynamically-linked programs and 64-bit programs.

See Also `exec(2)`, `fork(2)`, `pcsample(2)`, `times(2)`, `monitor(3C)`, `prof(5)`

Notes In Solaris releases prior to 2.6, calling `profil()` in a multithreaded program would impact only the calling LWP; the profile state was not inherited at LWP creation time. To profile a multithreaded program with a global profile buffer, each thread needed to issue a call to `profil()` at threads start-up time, and each thread had to be a bound thread. This was cumbersome and did not easily support dynamically turning profiling on and off. In Solaris 2.6, the `profil()` system call for multithreaded processes has global impact — that is, a call to `profil()` impacts all LWPs/threads in the process. This may cause applications that depend on the previous per-LWP semantic to break, but it is expected to improve multithreaded programs that wish to turn profiling on and off dynamically at runtime.

Name pset_bind – bind LWPs to a set of processors

Synopsis

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

int pset_bind(psetid_t pset, idtype_t idtype, id_t id, psetid_t *opset);
```

Description The `pset_bind()` function binds the LWP or set of LWPs specified by *idtype* and *id* to the processor set specified by *pset*. If *opset* is not `NULL`, `pset_bind()` sets the `psetid_t` variable pointed to by *opset* to the previous processor set binding of one of the specified LWP, or to `PS_NONE` if the selected LWP was not bound.

If *idtype* is `P_PID`, the binding affects all LWPs of the process with process ID (PID) *id*.

If *idtype* is `P_LWPID`, the binding affects the LWP of the current process with LWP ID *id*.

If *idtype* is `P_TASKID`, the binding affects all LWPs of all processes with task ID *id*.

If *idtype* is `P_PROJID`, the binding affects all LWPs of all processes with project ID *id*.

If *idtype* is `P_ZONEID`, the binding affects all LWPs of all processes with zone ID *id*.

If *idtype* is `P_CTID`, the binding affects all LWPs of all processes with process contract ID *id*.

If *id* is `P_MYID`, the specified LWP, process, task, process, zone, or process contract is the current one.

If *pset* is `PS_NONE`, the processor set bindings of the specified LWPs are cleared.

If *pset* is `PS_QUERY`, the processor set bindings are not changed.

If *pset* is `PS_MYID`, the specified LWPs are bound to the same processor set as the caller. If the caller is not bound to a processor set, the processor set bindings are cleared.

The {`PRIV_SYS_RES_CONFIG`} privilege must be asserted in the effective set of the calling process or *pset* must be `PS_QUERY`.

LWPs that have been bound to a processor with `processor_bind(2)` may also be bound to a processor set if the processor is part of the processor set. If this occurs, the binding to the processor remains in effect. If the processor binding is later removed, the processor set binding becomes effective.

Processor set bindings are inherited across `fork(2)` and `exec(2)`.

Return Values Upon successful completion, 0 is returned. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors The `pset_bind()` function will fail if:

EBUSY One of the LWPs is bound to a processor, and the specified processor set does not include that processor.

EFAULT The location pointed to by *opset* was not NULL and not writable by the user.

EINVAL An invalid processor set ID was specified; or *idtype* was not P_PID, P_LWPID, P_PROJID, P_TASKID, P_ZONEID, or P_CTID.

ENOTSUP The pools facility is active. See pooladm(1M) and pool_set_status(3POOL) for information about enabling and disabling the pools facility. Processes can be bound to pools using the poolbind(1M) utility or the pool_set_binding(3POOL) function.

Binding a system process to a processor set is not supported.

EPERM The {PRIV_PROC_OWNER} is not asserted in the effecive set of the calling process and either the real or effective user ID of the calling process does not match the real or effective user ID of one of the LWPs being bound, or the processor set from which one or more of the LWPs are being unbound has the PSET_NOESCAPE attribute set and {PRIV_SYS_RES_CONFIG) is not asserted in the effective set of the calling process. See pset_setattr(2) for more information about processor set attributes.

ESRCH No processes, LWPs, or tasks were found to match the criteria specified by *idtype* and *id*.

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe

See Also

pbind(1M), pooladm(1M), poolbind(1M), psrset(1M), exec(2), fork(2), processor_bind(2), pset_create(2), pset_info(2), pset_setattr(2), pool_set_binding(3POOL), pool_set_status(3POOL), pset_getloadavg(3C), process(4), project(4), attributes(5), privileges(5)

Name pset_create, pset_destroy, pset_assign – manage sets of processors

Synopsis

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

int pset_create(psetid_t *newpset);

int pset_destroy(psetid_t pset);

int pset_assign(psetid_t pset, processorid_t cpu, psetid_t *opset);
```

Description These functions control the creation and management of sets of processors. Processor sets allow a subset of the system's processors to be set aside for exclusive use by specified LWPs and processes. The binding of LWPs and processes to processor sets is controlled by `pset_bind(2)`.

The `pset_create()` function creates an empty processor set that contains no processors. On successful return, *newpset* will contain the ID of the new processor set.

The `pset_destroy()` function destroys the processor set *pset*, releasing its constituent processors and processes. If *pset* is `PS_MYID`, the processor set to which the caller is bound is destroyed.

The `pset_assign()` function assigns the processor *cpu* to the processor set *pset*. A processor that has been assigned to a processor set will run only LWPs and processes that have been explicitly bound to that processor set, unless another LWP requires a resource that is only available on that processor.

On successful return, if *opset* is non-null, *opset* will contain the processor set ID of the former processor set of the processor.

If *pset* is `PS_NONE`, `pset_assign()` releases processor *cpu* from its current processor set.

If *pset* is `PS_QUERY`, `pset_assign()` makes no change to processor sets, but returns the current processor set ID of processor *cpu* in *opset*.

If *pset* is `PS_MYID`, processor *cpu* is assigned to the processor set to which the caller belongs. If the caller does not belong to a processor set, processor *cpu* is released from its current processor set.

These functions are restricted to privileged processes, except for `pset_assign()` when *pset* is `PS_QUERY`.

Return Values Upon successful completion, these functions return `0`. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors These functions will fail if:

`EBUSY` The processor could not be moved to the specified processor set.

`EFAULT` The location pointed to by *newpset* was not writable by the user, or the location pointed to by *opset* was not `NULL` and not writable by the user.

EINVAL	The specified processor does not exist, the specified processor is not on-line, or an invalid processor set was specified.
ENOMEM	There was insufficient space for pset_create to create a new processor set.
ENOTSUP	The pools facility is active. See pooladm(1M) and pool_set_status(3POOL) for information about enabling and disabling the pools facility.
EPERM	The {PRIV_SYS_RES_CONFIG} privilege is not asserted in the effective set of the calling process.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe

See Also pooladm(1M), psradm(1M), psrinfo(1M), psrset(1M), p_online(2), processor_bind(2), pset_bind(2), pset_info(2), pool_set_status(3POOL), pset_getloadavg(3C), attributes(5), privileges(5)

Notes The processor set type of PS_SYSTEM is no longer supported.

Processors with LWPs bound to them using processor_bind(2) cannot be assigned to a new processor set. If this is attempted, pset_assign() will fail and set errno to EBUSY.

Name pset_info – get information about a processor set

Synopsis

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

int pset_info(psetid_t pset, int *type, uint_t *numcpus,
     processorid_t *cpulist);
```

Description The `pset_info()` function returns information on the processor set *pset*.

If *type* is non-null, then on successful completion the type of the processor set will be stored in the location pointed to by *type*. The only type supported for active processor sets is `PS_PRIVATE`.

If *numcpus* is non-null, then on successful completion the number of processors in the processor set will be stored in the location pointed to by *numcpus*.

If *numcpus* and *cpulist* are both non-null, then *cpulist* points to a buffer where a list of processors assigned to the processor set is to be stored, and *numcpus* points to the maximum number of processor IDs the buffer can hold. On successful completion, the list of processors up to the maximum buffer size is stored in the buffer pointed to by *cpulist*.

If *pset* is `PS_NONE`, the list of processors not assigned to any processor set will be stored in the buffer pointed to by *cpulist*, and the number of such processors will be stored in the location pointed to by *numcpus*. The location pointed to by *type* will be set to `PS_NONE`.

If *pset* is `PS_MYID`, the processor list and number of processors returned will be those of the processor set to which the caller is bound. If the caller is not bound to a processor set, the result will be equivalent to setting pset to `PS_NONE`.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `pset_info()` function will fail if:

`EFAULT` The location pointed to by *type*, *numcpus*, or *cpulist* was not null and not writable by the user.

`EINVAL` An invalid processor set ID was specified.

The caller is in a non-global zone, the pools facility is active, and the processor is not a member of the zone's pool's processor set.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Stable

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

See Also pooladm(1M), psrinfo(1M), psrset(1M), zoneadm(1M), processor_info(2), pset_assign(2), pset_bind(2), pset_create(2), pset_destroy(2), pset_getloadavg(3C), attributes(5)

Notes The processor set of type PS_SYSTEM is no longer supported.

Name pset_list – get list of processor sets

Synopsis

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

int pset_list(psetid_t *psetlist, uint_t *numpsets);
```

Description The `pset_list()` function returns a list of processor sets in the system.

If *numpsets* is non-null, then on successful completion the number of processor sets in the system will be stored in the location pointed to by *numpsets*.

If *numpsets* and *psetlist* are both non-null, then *psetlist* points to a buffer where a list of processor sets in the system is to be stored, and *numpsets* points to the maximum number of processor set IDs the buffer can hold. On successful completion, the list of processor sets up to the maximum buffer size is stored in the buffer pointed to by *psetlist*.

Return Values Upon successful completion, 0 is returned. Otherwise, -1 is returned and `errno` is set to indicate the error.

Errors The `pset_list()` function will fail if:

`EFAULT` The location pointed to by *psetlist* or *numpsets* was not null and not writable by the user.

Usage If the caller is in a non-global zone and the pools facility is active, `pset_list()` returns only the processor set of the pool to which the zone is bound.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Stable
MT-Level	Async-Signal-Safe

See Also `pooladm(1M)`, `psrset(1M)`, `zoneadm(1M)`, `processor_info(2)`, `pset_bind(2)`, `pset_create(2)`, `pset_info(2)`, `pset_getloadavg(3C)`, `attributes(5)`

Name pset_setattr, pset_getattr – set or get processor set attributes

Synopsis

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

int pset_setattr(psetid_t pset, uint_t attr);

int pset_getattr(psetid_t pset, uint_t *attr);
```

Description The `pset_setattr()` function sets attributes of the processor set specified by *pset*. The bitmask of attributes to be set or cleared is specified by *attr*.

The pset_getattr function returns attributes of the processor set specified by *pset*. On successful return, *attr* will contain the bitmask of attributes for the specified processor set.

The value of the *attr* argument is the bitwise inclusive-OR of these attributes, defined in `<sys/pset.h>`:

`PSET_NOESCAPE` Unbinding of LWPs from the processor set with this attribute requires the {`PRIV_SYS_RES_CONFIG`} privilege to be asserted in the effective set of the calling process.

The binding of LWPs and processes to processor sets is controlled by `pset_bind(2)`. When the `PSET_NOESCAPE` attribute is cleared, a process calling `pset_bind()` can clear the processor set binding of any LWP whose real or effective user ID matches its own real of effective user ID. Setting `PSET_NOESCAPE` attribute forces `pset_bind()` to requires the {`PRIV_SYS_RES_CONFIG`} privilege to be asserted in the effective set of the calling process.

Return Values Upon successful completion, these functions return 0. Otherwise, -1 is returned and `errno` is set to indicate the error.

Errors These function will fail if:

`EFAULT` The location pointed to by *attr* was not writable by the user.

`EINVAL` An invalid processor set ID was specified.

The caller is in a non-global zone, the pools facility is active, and the processor is not a member of the zone's pool's processor set.

`ENOTSUP` The pools facility is active. See `pooladm(1M)` and `pool_set_status(3POOL)` for information about enabling and disabling the pools facility.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Stable
MT-Level	Async-Signal-Safe

See Also pooladm(1M), pooladm(1M), psrset(1M), zoneadm(1M), pset_bind(2), pool_set_status(3POOL), attributes(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 The `putmsg()` function 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 `putpmsg()` function 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()`.

The *fildes* argument specifies a file descriptor referencing an open stream. The *ctlptr* and *dataptr* arguments 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 */
```

The *ctlptr* argument points to the structure describing the control part, if any, to be included in the message. The `buf` member in the `strbuf` structure points to the buffer where the control information resides, and the `len` member indicates the number of bytes to be sent. The `maxlen` member is not used in `putmsg()` (see `getmsg(2)`). In a similar manner, *dataptr* specifies the data, if any, to be included in the message. The *flags* argument 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` member 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` member 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. The *flags* argument 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.

The putmsg() or putpmsg() function 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, 0 is returned. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The putmsg() and putpmsg() functions will fail if:

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	The *fildes* argument is not a valid file descriptor open for writing.
EFAULT	The *ctlptr* or *dataptr* argument points to an illegal address.
EINTR	A signal was caught during the execution of the putmsg() function.
EINVAL	An undefined value was specified in *flags*; *flags* is set to RS_HIPRI and no control part was supplied; or the stream referenced by *fildes* is linked below a multiplexor.
ENOSR	Buffers could not be allocated for the message that was to be created due to insufficient streams memory resources.
ENOSTR	The *fildes* argument 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.
EPIPE or EIO	The *fildes* argument refers to a streams-based pipe and the other end of the pipe is closed. A SIGPIPE signal is generated for the calling thread. This error condition occurs only with SUS-conforming applications. See standards(5).
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.

In addition, `putmsg()` and `putpmsg()` will fail if the stream head had processed an asynchronous error before the call. In this case, the value of `errno` does not reflect the result of `putmsg()` or `putpmsg()` but reflects the prior error.

The `putpmsg()` function will fail if:

`EINVAL` The *flags* argument is set to `MSG_HIPRI` and *band* is non-zero.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also

`Intro(2)`, `getmsg(2)`, `poll(2)`, `read(2)`, `write(2)`, `attributes(5)`, `standards(5)`

STREAMS Programming Guide

Name read, readv, pread – read from file

Synopsis

```
#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);

#include <sys/uio.h>

ssize_t readv(int fildes, const struct iovec *iov, int iovcnt);
```

Description The `read()` function attempts to read *nbyte* bytes from the file associated with the open file descriptor, *fildes*, into the buffer pointed to by *buf*.

If *nbyte* is 0, `read()` returns `0` and has no other results.

On files that support seeking (for example, a regular file), the `read()` starts at a position in the file given by the file offset associated with *fildes*. The file offset is incremented by the number of bytes actually read.

Files that do not support seeking (for example, terminals) always read from the current position. The value of a file offset associated with such a file is undefined.

If *fildes* refers to a socket, `read()` is equivalent to recv(3SOCKET) with no flags set.

No data transfer will occur past the current end-of-file. If the starting position is at or after the end-of-file, `0` will be returned. If the file refers to a device special file, the result of subsequent `read()` requests is implementation-dependent.

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 socket or a stream that is not a pipe, a FIFO, or a terminal, and the file 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.

The read() function reads data previously written to a file. If any portion of a regular file prior to the end-of-file has not been written, read() returns bytes with value 0. For example, lseek(2) allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return bytes with value 0 until data is written into the gap.

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with *fildes*.

Upon successful completion, where *nbyte* is greater than 0, read() will mark for update the st_atime field of the file, and return the number of bytes read. This number will never be greater than *nbyte*. The value returned may be less than *nbyte* if the number of bytes left in the file is less than *nbyte*, if the read() request was interrupted by a signal, or if the file is a pipe or FIFO or special file and has fewer than *nbyte* bytes immediately available for reading. For example, a read() from a file associated with a terminal may return one typed line of data.

If a read() is interrupted by a signal before it reads any data, it will return −1 with errno set to EINTR.

If a read() is interrupted by a signal after it has successfully read some data, it will return the number of bytes read.

A read() from a streams file can read data 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, and can be tested with the I_GRDOPT ioctl(). In byte-stream mode, read() retrieves data from the stream until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In streams message-nondiscard mode, read() retrieves data until as many bytes as were requested are transferred, or until a message boundary is reached. If read() does not retrieve all the data in a message, the remaining data is left on the stream, and can be retrieved by the next read() call. Message-discard mode also retrieves data until as many bytes as were requested are transferred, or a message boundary is reached. However, unread data remaining in a message after the read() returns is discarded, and is not available for a subsequent read(), readv() or getmsg(2) call.

How `read()` handles zero-byte streams 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. The `read()` function then returns the number of bytes read, and places the zero-byte message back on the stream to be retrieved by the next `read()`, `readv()` or `getmsg(2)`. In message-nondiscard mode or message-discard mode, a zero-byte message returns `0` and the message is removed from the stream. When a zero-byte message is read as the first message on a stream, the message is removed from the stream and `0` is returned, regardless of the read mode.

A `read()` 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.

By default, streams are in control-normal mode, in which a `read()` from a streams file can only process messages that contain a data part but do not contain a control part. The `read()` fails if a message containing a control part 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()` command. In control-data mode, `read()` converts any control part to data and passes it to the application before passing any data part originally present in the same message. In control-discard mode, `read()` discards message control parts but returns to the process any data part in the message.

In addition, `read()` and `readv()` will fail if the stream head had processed an asynchronous error before the call. In this case, the value of `errno` does not reflect the result of `read()` or `readv()` but reflects the prior error. 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`.

`readv()` The `readv()` function is equivalent to `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 *iovcnt* argument is valid if greater than 0 and less than or equal to {`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 where data should be placed. The `readv()` function always fills an area completely before proceeding to the next.

Upon successful completion, `readv()` marks for update the `st_atime` field of the file.

`pread()` The `pread()` function 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. `pread()` will read up to the maximum offset value that can be represented in an `off_t` for regular files. An attempt to perform a `pread()` on a file that is incapable of seeking results in an error.

Return Values Upon successful completion, `read()` and `readv()` return a non-negative integer indicating the number of bytes actually read. Otherwise, the functions return −1 and set `errno` to indicate the error.

Errors The `read()`, `readv()`, and `pread()` functions will fail if:

`EAGAIN` Mandatory file/record locking was set, `O_NDELAY` or `O_NONBLOCK` was set, and there was a blocking record lock; total amount of system memory available when reading using raw I/O is temporarily insufficient; no data is waiting to be read on a file associated with a tty device and `O_NONBLOCK` was set; or no message is waiting to be read on a stream and `O_NDELAY` or `O_NONBLOCK` was set.

`EBADF` The *fildes* argument 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.

`EINTR` A signal was caught during the read operation and no data was transferred.

`EINVAL` An attempt was made 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` The *fildes* argument 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` The *fildes* argument 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.

The `read()` and `pread()` functions will fail if:

`EFAULT` The *buf* argument points to an illegal address.

`EINVAL` The *nbyte* argument overflowed an `ssize_t`.

The `read()` and `readv()` functions will fail if:

`EOVERFLOW` The file is a regular file, *nbyte* is greater than 0, the starting position is before the end-of-file, and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fildes*.

The `readv()` function may fail if:

EFAULT The *iov* argument points outside the allocated address space.

EINVAL The *iovcnt* argument was less than or equal to 0 or greater than {IOV_MAX}. See Intro(2) for a definition of {IOV_MAX}).

One of the iov_len values in the *iov* array was negative, or the sum of the iov_len values in the *iov* array overflowed an ssize_t.

The pread() function will fail and the file pointer remain unchanged if:

ESPIPE The *fildes* argument is associated with a pipe or FIFO.

Usage The pread() function has a transitional interface for 64-bit file offsets. See lf64(5).

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	read() is Async-Signal-Safe
Standard	See standards(5).

See Also Intro(2), chmod(2), creat(2), dup(2), fcntl(2), getmsg(2), ioctl(2), lseek(2), open(2), pipe(2), recv(3SOCKET), attributes(5), lf64(5), standards(5), streamio(7I), termio(7I)

Name readlink – read the contents of a symbolic link

Synopsis

```
#include <unistd.h>

ssize_t readlink(const char *restrict path,
     char *restrict buf, size_t bufsiz);
```

Description The `readlink()` function places the contents of the symbolic link referred to by *path* in the buffer *buf* which has size *bufsiz*. If the number of bytes in the symbolic link is less than *bufsiz*, the contents of the remainder of *buf* are left unchanged. If the *buf* argument is not large enough to contain the link content, the first *bufsize* bytes are placed in *buf*.

Return Values Upon successful completion, `readlink()` returns the count of bytes placed in the buffer. Otherwise, it returns −1, leaves the buffer unchanged, and sets `errno` to indicate the error.

Errors The `readlink()` function will fail 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 *path* argument names a file that is not a symbolic link.
`EIO`	An I/O error occurred while reading from the file system.
`ENOENT`	A component of *path* does not name an existing file or *path* is an empty string.
`ELOOP`	A loop exists in symbolic links encountered during resolution of the *path* argument.
`ENAMETOOLONG`	The length of *path* exceeds {`PATH_MAX`}, or a pathname component is longer than {`NAME_MAX`} while `_POSIX_NO_TRUNC` is in effect.
`ENOTDIR`	A component of the path prefix is not a directory.
`ENOSYS`	The file system does not support symbolic links.

The `readlink()` function may fail if:

`EACCES`	Read permission is denied for the directory.
`ELOOP`	More than {`SYMLOOP_MAX`} symbolic links were encountered in resolving *path*.
`ENAMETOOLONG`	As a result of encountering a symbolic link in resolution of the path argument, the length of the substituted pathname string exceeded {`PATH_MAX`}.

Usage Portable applications should not assume that the returned contents of the symbolic link are null-terminated.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also stat(2), symlink(2), attributes(5), standards(5)

Name rename, renameat – change the name of a file

Synopsis

```
#include <stdio.h>

int rename(const char *old, const char *new);

#include <unistd.h>

int renameat(int fromfd, const char *old, int tofd,
     const char *new);
```

XPG3

```
#include <unistd.h>

int rename(const char *old, const char *new);
```

Description The `rename()` function changes the name of a file. The *old* argument points to the pathname of the file to be renamed. The *new* argument points to the new path name of the file.

The `renameat()` function renames an entry in a directory, possibly moving the entry into a different directory. See `fsattr(5)`. If the *old* argument is an absolute path, the *fromfd* is ignored. Otherwise it is resolved relative to the *fromfd* argument rather than the current working directory. Similarly, if the *new* argument is not absolute, it is resolved relative to the *tofd* argument. If either *fromfd* or *tofd* have the value `AT_FDCWD`, defined in `<fcntl.h>`, and their respective paths are relative, the path is resolved relative to the current working directory.

Current implementation restrictions will cause the `renameat()` function to return an error if an attempt is made to rename an extended attribute file to a regular (non-attribute) file, or to rename a regular file to an extended attribute file.

If *old* and *new* both refer to the same existing file, the `rename()` and `renameat()` functions return 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()` or `renameat()` 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()` and `renameat()` functions will mark for update the `st_ctime` and `st_mtime` fields of the parent directory of each file.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate an error.

Errors The `rename()` function will fail if:

`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`	The *new* argument is a directory and the mount point for a mounted file system.
`EDQUOT`	The directory where the new name entry is being placed cannot be extended because the user's quota of disk blocks on that file system has been exhausted.
`EEXIST`	The link named by *new* is a directory containing entries other than '`.`' (the directory itself) and '`..`' (the parent directory).
`EFAULT`	Either *old* or *new* references an invalid address.
`EILSEQ`	The path argument includes non-UTF8 characters and the file system accepts only file names where all characters are part of the UTF-8 character codeset.

EINVAL	The *new* argument directory pathname contains a path prefix that names the *old* directory, or an attempt was made to rename a regular file to an extended attribute or from an extended attribute to a regular file.
EIO	An I/O error occurred while making or updating a directory entry.
EISDIR	The *new* argument 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 name an existing file, a component of the path prefix of *new* 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 file that is not a directory, or *tofd* and *dirfd* in renameat() do not reference a directory.
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.

The renameat() functions will fail if:

ENOTSUP	An attempt was made to rename a regular file as an attribute file or to rename an attribute file as a regular file.

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	For rename(), see standards(5).

See Also

chmod(2), link(2), unlink(2), attributes(5), fsattr(5), standards(5)

Notes The system can deadlock if there is a loop in the file system graph. Such a loop can occur if there is an entry in directory `a`, `a/name1`, that is a hard link to directory `b`, and an entry in directory `b`, `b/name2`, that is a hard link to directory `a`. When such a loop exists and two separate processes attempt to rename `a/name1` to `b/name2` and `b/name2` to `a/name1`, the system may deadlock attempting to lock both directories for modification. Use symbolic links instead of hard links for directories.

Name resolvepath – resolve all symbolic links of a path name

Synopsis

```
#include <unistd.h>

int resolvepath(const char *path, char *buf, size_t bufsiz);
```

Description The `resolvepath()` function fully resolves all symbolic links in the path name *path* into a resulting path name free of symbolic links and places the resulting path name in the buffer *buf* which has size *bufsiz*. The resulting path name names the same file or directory as the original path name. All "." components are eliminated and every non-leading ". ." component is eliminated together with its preceding directory component. If leading ". ." components reach to the root directory, they are replaced by "/". If the number of bytes in the resulting path name is less than *bufsiz*, the contents of the remainder of *buf* are unspecified.

Return Values Upon successful completion, `resolvepath()` returns the count of bytes placed in the buffer. Otherwise, it returns −1, leaves the buffer unchanged, and sets `errno` to indicate the error.

Errors The `resolvepath()` function will fail if:

`EACCES`	Search permission is denied for a component of the path prefix of *path* or for a path prefix component resulting from the resolution of a symbolic link.
`EFAULT`	The *path* or *buf* argument points to an illegal address.
`EIO`	An I/O error occurred while reading from the file system.
`ENOENT`	The *path* argument is an empty string or a component of *path* or a path name component produced by resolving a symbolic link does not name an existing file.
`ELOOP`	Too many symbolic links were encountered in resolving *path*.
`ENAMETOOLONG`	The length of *path* exceeds `PATH_MAX`, or a path name component is longer than `NAME_MAX`. Path name resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX` or a component whose length exceeds `NAME_MAX`.
`ENOTDIR`	A component of the path prefix of *path* or of a path prefix component resulting from the resolution of a symbolic link is not a directory.

Usage No more than `PATH_MAX` bytes will be placed in the buffer. Applications should not assume that the returned contents of the buffer are null-terminated.

See Also `readlink(2)`, `realpath(3C)`

Name rmdir – remove a directory

Synopsis

```
#include <unistd.h>

int rmdir(const char *path);
```

Description The `rmdir()` function 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, `0` is returned. Otherwise, `−1` is returned, `errno` is set to indicate the error, and the named directory is not changed.

Errors The `rmdir()` function will fail if:

`EACCES` Search permission is denied for a component of the path prefix and {`PRIV_FILE_DAC_SEARCH`} is not asserted in the effective set of the calling process

Write permission is denied on the directory containing the directory to be removed and {`PRIV_FILE_DAC_WRITE`} is not asserted.

The parent directory has the `S_ISVTX` variable set, is not owned by the user, and {`PRIV_FILE_OWNER`} is not asserted.

The directory is not owned by the user and is not writable by the 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` The *path* argument points to an illegal address.

`EINVAL` The directory to be removed is the current directory, or the final component of *path* is ".".

`EILSEQ` The path argument includes non-UTF8 characters and the file system accepts only file names where all characters are part of the UTF-8 character codeset.

`EIO` An I/O error occurred while accessing 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	The named directory does not exist or is the null pathname.
ENOLINK	The *path* argument 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.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also mkdir(1), rm(1), mkdir(2), attributes(5), privileges(5), standards(5)

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 The `semctl()` function 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_t        *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 `semvals` into array pointed to by *arg*.`array`. {READ}

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

The following 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    /* access permission bits only */
```

This command can be executed only by a process that has either the {`PRIV_IPC_OWNER`} privilege or an effective user ID equal to the value of `msg_perm.cuid` or `msg_perm.uid` in the data structure associated with *msqid*. Only a process with the {`PRIV_SYS_IPC_CONFIG`} privilege can raise the value of `msg_qbytes`.

`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 be executed only by a process that has the {`PRIV_IPC_OWNER`} privilege or an effective user ID equal 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 The `semctl()` function will fail if:

`EACCES` Operation permission is denied to the calling process (see `Intro(2)`).

`EFAULT` The source or target is not a valid address in the user process.

`EINVAL` The *semid* argument is not a valid semaphore identifier; the *semnum* argument is less than 0 or greater than `sem_nsems` −1; or the *cmd* argument is not a valid command or is `IPC_SET` and `sem_perm.uid` or `sem_perm.gid` is not valid.

`EPERM` The *cmd* argument is equal to `IPC_RMID` or `IPC_SET`, the effective user ID of the calling process is not equal to the value of `sem_perm.cuid` or `sem_perm.uid` in

the data structure associated with *semid*, and {PRIV_IPC_OWNER} is not asserted in the effective set of the calling process.

EOVERFLOW — The *cmd* argument is IPC_STAT and *uid* or *gid* is too large to be stored in the structure pointed to by *arg.buf*.

ERANGE — The *cmd* argument is SETVAL or SETALL and the value to which semval is to be set is greater than the system imposed maximum.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also ipcs(1), Intro(2), semget(2), semop(2), attributes(5), privileges(5), standards(5)

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 The `semget()` function 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 representing a semaphore identifier is returned. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors The `semget()` function will fail if:

`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` The *nsems* argument is either less than or equal to 0 or greater than the system-imposed limit. See NOTES.

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 0.

`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 semaphores or semaphore identifiers system-wide would be exceeded. See NOTES.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also ipcrm(1), ipcs(1), rctladm(1M), Intro(2), semctl(2), semop(2), setrctl(2), ftok(3C), attributes(5), standards(5)

Notes The system-imposed limit on the value of the *nsems* argument is the maintained on a per-process basis using the `process.max-sem-nsems` resource control.

The system-imposed limit on the number of semaphore identifiers is maintained on a per-project basis using the `project.max-sem-ids` resource control. The `zone.max-sem-ids` resource control limis the total number of semaphore identifiers that can be allocated by a zone.

See rctladm(1M) and setrctl(2) for information about using resource controls.

Name semids – discover all semaphore identifiers

Synopsis

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

int semids(int *buf, uint_t nids, uint_t *pnids);
```

Description The `semids()` function copies all active semaphore identifiers from the system into the user-defined buffer specified by *buf*, provided that the number of such identifiers is not greater than the number of integers the buffer can contain, as specified by *nids*. If the size of the buffer is insufficient to contain all of the active semaphore identifiers in the system, none are copied.

Whether or not the size of the buffer is sufficient to contain all of them, the number of active semaphore identifiers in the system is copied into the unsigned integer pointed to by *pnids*.

If *nids* is 0 or less than the number of active semaphore identifiers in the system, *buf* is ignored.

Return Values Upon successful completion, `semids()` returns `0`. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `semids()` function will fail if:

`EFAULT` The *buf* or *pnids* argument points to an illegal address.

Usage The `semids()` function returns a snapshot of all the active semaphore identifiers in the system. More may be added and some may be removed before they can be used by the caller.

Examples **EXAMPLE 1** `semids()` example

This is sample C code indicating how to use the `semids()` function.

```
void
examine_semids()
{
     int *ids = NULL;
     uint_t nids = 0;
     uint_t n;
     int i;

     for (;;) {
          if (semids(ids, nids, &n) != 0) {
               perror("semids");
               exit(1);
          }
          if (n <= nids)     /* we got them all */
               break;
          /* we need a bigger buffer */
          ids = realloc(ids, (nids = n) * sizeof (int));
     }

     for (i = 0; i < n; i++)
```

EXAMPLE 1 `semids()` example *(Continued)*

```
            process_semid(ids[i]);

        free(ids);
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

See Also

`ipcrm(1)`, `ipcs(1)`, `Intro(2)`, `semctl(2)`, `semget(2)`, `semop(2)`, `attributes(5)`

Name semop, semtimedop – 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);

int semtimedop(int semid, struct sembuf *sops, size_t nsops,
     const struct timespec *timeout);
```

Description The `semop()` function is used to perform atomically an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by *semid*. The *sops* argument is a pointer to the array of semaphore-operation structures. The *nsops* argument is the number of such structures in the array.

Each `sembuf` structure contains 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.

A process maintains a value, `semadj`, for each semaphore it modifies. This value contains the cumulative effect of operations the process has performed on an individual semaphore with the `SEM_UNDO` flag set (so that they can be undone if the process terminates unexpectedly). The value of `semadj` can affect the behavior of calls to `semop()`, `semtimedop()`, `exit()`, and `_exit()` (the latter two functions documented on `exit(2)`), but is otherwise unobservable. See below for details.

The `sem_op` member specifies one of three semaphore operations:

1. The `sem_op` member 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 thread until one of the following conditions occur:
  - The value of `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 thread is awaiting action is removed from the system (see `semctl(2)`). When this occurs, `errno` is set to `EIDRM` and −1 is returned.
  - The calling thread 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 thread resumes execution in the manner prescribed in `sigaction(2)`.

2. The `sem_op` member 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.

3. The `sem_op` member is 0; {READ}
   - If `semval` is 0, `semop()` returns immediately.
   - If `semval` is not equal to 0 and (*sem_flg*&`IPC_NOWAIT`) is true, `semop()` returns immediately.
   - If `semval` is not equal to 0 and (*sem_flg*&`IPC_NOWAIT`) is false, `semop()` increments the `semzcnt` associated with the specified semaphore and suspends execution of the calling thread until one of the following occurs:
     - The value of `semval` becomes 0, at which time the value of `semzcnt` associated with the specified semaphore is set to 0 and all processes waiting on `semval` to become 0 are awakened.
     - The *semid* for which the calling thread is awaiting action is removed from the system. When this occurs, `errno` is set to `EIDRM` and −1 is returned.
     - The calling thread 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 thread resumes execution in the manner prescribed in `sigaction(2)`.

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

The `semtimedop()` function behaves as `semop()` except when it must suspend execution of the calling process to complete its operation. If `semtimedop()` must suspend the calling process after the time interval specified in *timeout* expires, or if the timeout expires while the process is suspended, `semtimedop()` returns with an error. If the `timespec` structure pointed to by

timeout is zero-valued and `semtimedop()` needs to suspend the calling process to complete the requested operation(s), it returns immediately with an error. If *timeout* is the `NULL` pointer, the behavior of `semtimedop()` is identical to that of `semop()`.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `semop()` and `semtimedop()` functions will fail if:

`E2BIG` The *nsops* argument is greater than the system-imposed maximum. See NOTES.

`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` The *sops* argument points to an illegal address.

`EFBIG` The value of `sem_num` is less than 0 or greater than or equal to the number of semaphores in the set associated with *semid*.

`EIDRM` A *semid* was removed from the system.

`EINTR` A signal was received.

`EINVAL` The *semid* argument is not a valid semaphore identifier, or the number of individual semaphores for which the calling process requests a `SEM_UNDO` operation would exceed the system-imposed limit. Solaris does not impose a limit on the number of individual semaphores for which the calling process requests a `SEM_UNDO` operation.

`ENOSPC` The limit on the number of individual processes requesting a `SEM_UNDO` operation would be exceeded. Solaris does not impose a limit on the number of individual processes requesting an `SEM_UNDO` operation.

`ERANGE` An operation would cause a `semval` or a `semadj` value to overflow the system-imposed limit.

The `semtimedop()` function will fail if:

`EAGAIN` The timeout expired before the requested operation could be completed.

The `semtimedop()` function will fail if one of the following is detected:

`EFAULT` The *timeout* argument points to an illegal address.

`EINVAL` The *timeout* argument specified a `tv_sec` or `tv_nsec` value less than 0, or a `tv_nsec` value greater than or equal to 1000 million.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	`semop()` is Standard.

See Also `ipcs(1)`, `rctladm(1M)`, `Intro(2)`, `exec(2)`, `exit(2)`, `fork(2)`, `semctl(2)`, `semget(2)`, `setrctl(2)`, `sigaction(2)`, `attributes(5)`, `standards(5)`

Notes The system-imposed maximum on *nsops* for a semaphore identifier is the minimum enforced value of the `process.max-sem-ops` resource control of the creating process at the time `semget(2)` was used to allocate the identifier.

See `rctladm(1M)` and `setrctl(2)` for information about using resource controls.

Name setpgid – set process group ID

Synopsis

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

int setpgid(pid_t pid, pid_t pgid);
```

Description The `setpgid()` function 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, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `setpgid()` function will fail if:

`EACCES` The *pid* argument matches the process ID of a child process of the calling process and the child process has successfully executed one of the *exec* family of functions (see `exec(2)`).

`EINVAL` The *pgid* argument is less than (`pid_t`) `0` or greater than or equal to `PID_MAX`, or 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` The *pid* argument 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` The *pgid* argument 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` The *pid* argument does not match the process ID of the calling process or of a child process of the calling process.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also Intro(2), exec(2), exit(2), fork(2), getpid(2), getsid(2), attributes(5), standards(5)

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, the `setpgrp()` function 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 The `setpgrp()` function returns the value of the new process group ID.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also `setpgrp(1)`, `Intro(2)`, `exec(2)`, `fork(2)`, `getpid(2)`, `getsid(2)`, `kill(2)`, `signal(3C)`, `attributes(5)`, `standards(5)`

Name setrctl, getrctl – set or get resource control values

Synopsis

```
#include <rctl.h>

int setrctl(const char *controlname, rctlblk_t *old_blk,
     rctlblk_t *new_blk, uint_t flags);

int getrctl(const char *controlname, rctlblk_t *old_blk,
     rctlblk_t *new_blk, uint_t flags);
```

Description The `setrctl()` and `getrctl()` functions provide interfaces for the modification and retrieval of resource control (rctl) values on active entities on the system, such as processes, tasks, or projects. All resource controls are unsigned 64-bit integers; however, a collection of flags are defined that modify which rctl value is to be set or retrieved.

Resource controls are restricted to three levels: basic controls that can be modified by the owner of the calling process, privileged controls that can be modified only by privileged callers, and system controls that are fixed for the duration of the operating system instance. Setting or retrieving each of these controls is performed by setting the privilege field of the resource control block to `RCTL_BASIC`, `RCTL_PRIVILEGED`, or `RCTL_SYSTEM` with `rctlblk_set_privilege()` (see `rctlblk_set_value(3C)`).

For limits on collective entities such as the task or project, the process ID of the calling process is associated with the resource control value. This ID is available by using `rctlblk_get_recipient_pid()` (see `rctlblk_set_value(3C)`). These values are visible only to that process and privileged processes within the collective.

The `getrctl()` function provides a mechanism for iterating through all of the established values on a resource control. The iteration is primed by calling `getrctl()` with *old_blk* set to `NULL`, a valid resource control block pointer in *new_blk*, and specifying `RCTL_FIRST` in the *flags* argument. Once a resource control block has been obtained, repeated calls to `getrctl()` with `RCTL_NEXT` in the *flags* argument and the obtained control in the *old_blk* argument will return the next resource control block in the sequence. The iteration reports the end of the sequence by failing and setting `errno` to `ENOENT`.

The `getrctl()` function allows the calling process to get the current usage of a controlled resource using `RCTL_USAGE` as the *flags* value. The current value of the resource usage is placed in the value field of the resource control block specified by *new_blk*. This value is obtained with `rctlblk_set_value(3C)`. All other members of the returned block are undefined and might be invalid.

The `setrctl()` function allows the creation, modification, or deletion of action-value pairs on a given resource control. When passed `RCTL_INSERT` as the *flags* value, `setrctl()` expects *new_blk* to contain a new action-value pair for insertion into the sequence. For `RCTL_DELETE`, the block indicated by *new_blk* is deleted from the sequence. For `RCTL_REPLACE`, the block matching *old_blk* is deleted and replaced by the block indicated by *new_blk*. When (*flags* & `RCTL_USE_RECIPIENT_PID`) is non-zero, `setrctl()` uses the process ID set by

rctlblk_set_value(3C) when selecting the rctl value to insert, delete, or replace basic rctls. Otherwise, the process ID of the calling process is used.

The kernel maintains a history of which resource control values have triggered for a particular entity, retrievable from a resource control block with the rctlblk_set_value(3C) function. The insertion or deletion of a resource control value at or below the currently enforced value might cause the currently enforced value to be reset. In the case of insertion, the newly inserted value becomes the actively enforced value. All higher values that have previously triggered will have their firing times zeroed. In the case of deletion of the currently enforced value, the next higher value becomes the actively enforced value.

The various resource control block properties are described on the rctlblk_set_value(3C) manual page.

Resource controls are inherited from the predecessor process or task. One of the exec(2) functions can modify the resource controls of a process by resetting their histories, as noted above for insertion or deletion operations.

Return Values Upon successful completion, the `setrctl()` and `getrctl()` functions return 0. Otherwise they return −1 and set `errno` to indicate the error.

Errors The `setrctl()` and `getrctl()` functions will fail if:

`EFAULT` The *controlname*, *old_blk*, or *new_blk* argument points to an illegal address.

`EINVAL` No resource control with the given name is known to the system, or the resource control block contains properties that are not valid for the resource control specified.

`RCTL_USE_RECIPIENT_PID` was used to set a process scope rctl and the process ID set by rctlblk_set_value(3C) does not match the process ID of calling process.

`ENOENT` No value beyond the given resource control block exists.

`RCTL_USE_RECIPIENT_PID` was used and the process ID set by rctlblk_set_value(3C) does not exist within the current task, project, or zone, depending on the resource control name.

`ESRCH` No value matching the given resource control block was found for any of `RCTL_NEXT`, `RCTL_DELETE`, or `RCTL_REPLACE`.

`ENOTSUPP` The resource control requested by `RCTL_USAGE` does not support the usage operation.

The `setrctl()` function will fail if:

`EACCES` The rctl value specified cannot be changed by the current process, including the case where the recipient process ID does not match the calling process and the calling process is unprivileged.

EPERM An attempt to set a system limit was attempted.

Examples

EXAMPLE 1 Retrieve a rctl value.

Obtain the lowest enforced rctl value on the rctl limiting the number of LWPs in a task.

```
#include <rctl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

...

rctlblk_t *rblk;

if ((rblk = (rctlblk_t *)malloc(rctlblk_size())) == NULL) {
        (void) fprintf(stderr, "malloc failed: %s\n",
            strerror(errno));
        exit(1);
}

if (getrctl("task.max-lwps", NULL, rblk, RCTL_FIRST) == -1)
        (void) fprintf(stderr, "failed to get rctl: %s\n",
            strerror(errno));
else
        (void) printf("task.max-lwps = %llu\n",
            rctlblk_get_value(rblk));
```

Usage

Resource control blocks are matched on the value and privilege fields. Resource control operations act on the first matching resource control block. Duplicate resource control blocks are not permitted. Multiple blocks of equal value and privilege need to be entirely deleted and reinserted, rather than replaced, to have the correct outcome. Resource control blocks are sorted such that all blocks with the same value that lack the `RCTL_LOCAL_DENY` flag precede those having that flag set.

Only one `RCPRIV_BASIC` resource control value is permitted per process per control. Insertion of an `RCPRIV_BASIC` value will cause any existing `RCPRIV_BASIC` value owned by that process on the control to be deleted.

The resource control facility provides the backend implementation for both `setrctl()`/`getrctl()` and `setrlimit()`/`getrlimit()`. The facility behaves consistently when either of these interfaces is used exclusively; when using both interfaces, the caller must be aware of the ordering issues above, as well as the limit equivalencies described in the following paragraph.

The hard and soft process limits made available with `setrlimit()` and `getrlimit()` are mapped to the resource controls implementation. (New process resource controls will not be made available with the rlimit interface.) Because of the `RCTL_INSERT` and `RCTL_DELETE` operations, it is possible that the set of values defined on a resource control has more or fewer than the two values defined for an rlimit. In this case, the soft limit is the lowest priority resource control value with the `RCTL_LOCAL_DENY` flag set, and the hard limit is the resource control value with the lowest priority equal to or exceeding `RCPRIV_PRIVILEGED` with the `RCTL_LOCAL_DENY` flag set. If no identifiable soft limit exists on the resource control and `setrlimit()` is called, a new resource control value is created. If a resource control does not have the global `RCTL_GLOBAL_LOWERABLE` property set, its hard limit will not allow lowering by unprivileged callers.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

See Also `rctladm(1M)`, `getrlimit(2)`, `errno(3C)`, `rctlblk_set_value(3C)`, `attributes(5)`, `resource_controls(5)`

Name setregid – set real and effective group IDs

Synopsis

```
#include <unistd.h>

int setregid(gid_t rgid, gid_t egid);
```

Description The `setregid()` function is used to set the real and effective group IDs of the calling process. If *rgid* is −1, the real group ID is not changed; if *egid* is −1, the effective group ID is not changed. The real and effective group IDs may be set to different values in the same call.

If the {`PRIV_PROC_SETID`} privilege is asserted in the effective set of the calling process, the real group ID and the effective group ID can be set to any legal value.

If the {`PRIV_PROC_SETID`} privilege is not asserted in the effective set of the calling process, either the real group ID can be set to the saved set-group-ID from `execve(2)`, or the effective group ID can either be set to the saved set-group-ID or the real group ID.

In either case, if the real group ID is being changed (that is, if *rgid* is not −1), or the effective group ID is being changed to a value not equal to the real group ID, the saved set-group-ID is set equal to the new effective group ID.

Return Values Upon successful completion, 0 is returned. Otherwise, −1 is returned, `errno` is set to indicate the error, and neither of the group IDs will be changed.

Errors The `setregid()` function will fail if:

`EINVAL` The value of *rgid* or *egid* is less than 0 or greater than `UID_MAX` (defined in `<limits.h>`).

`EPERM` The {`PRIV_PROC_SETID`} privilege is not asserted in the effective set of the calling processes and a change was specified other than changing the real group ID to the saved set-group-ID or changing the effective group ID to the real group ID or the saved group ID.

Usage If a set-group-ID process sets its effective group ID to its real group ID, it can still set its effective group ID back to the saved set-group-ID.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also `execve(2)`, `getgid(2)`, `setreuid(2)`, `setuid(2)`, `attributes(5)`, `privileges(5)`, `standards(5)`

Name setreuid – set real and effective user IDs

Synopsis

```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);
```

Description The `setreuid()` function 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 {`PRIV_PROC_SETID`} privilege is asserted in the effective set of the calling process, the real user ID and the effective user ID can be set to any legal value.

If the {`PRIV_PROC_SETID`} privilege is not asserted in the effective set of the calling process, 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()` (see`exec(2)`) or the real 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.

All privileges are required to change to uid 0.

Return Values Upon successful completion, 0 is returned. Otherwise, −1 is returned, `errno` is set to indicate the error, and neither of the user IDs will be changed.

Errors The `setreuid()` function will fail if:

`EINVAL` The value of *ruid* or *euid* is less than `0` or greater than `UID_MAX` (defined in `<limits.h>`).

`EPERM` The {PRIV_PROC_SETID} privilege is not asserted in the effective set of the calling processes and a change was specified 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. See `privileges(5)` for additional restrictions which apply when changing to UID 0.

Usage If a set-user-ID 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.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also exec(2), getuid(2), setregid(2), setuid(2), attributes(5), privileges(5), standards(5)

Name setsid – create session and set process group ID

Synopsis

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

pid_t setsid(void);
```

Description The `setsid()` function creates a new session, if the calling process is not a process group leader. Upon return the calling process will be the session leader of this new session, will be the process group leader of a new process group, and will have no controlling terminal. The process group ID of the calling process will be set equal to the process ID of the calling process. The calling process will be the only process in the new process group and the only process in the new session.

Return Values Upon successful completion, `setsid()` returns the value of the process group ID of the calling process. Otherwise it returns `(pid_t)−1` and sets `errno` to indicate the error.

Errors The `setsid()` function will fail if:

`EPERM` The calling process is already a process group leader, or the process group ID of a process other than the calling process matches the process ID of the calling process.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `getsid(2)`, `setpgid(2)`, `setpgrp(2)`, `attributes(5)`, `standards(5)`

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 settaskid, gettaskid, getprojid – set or get task or project IDs

Synopsis

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

taskid_t settaskid(projid_t project, int flags);

taskid_t gettaskid(void);

#include <sys/types.h>
#include <sys/task.h>
#include <unistd.h>
#include <project.h>

projid_t getprojid(void);
```

Description The `settaskid()` function makes a request of the system to assign a new task ID to the calling process, changing the associated project ID to that specified. The calling process must have sufficient privileges to perform this operation. The *flags* argument should be either `TASK_NORMAL` for a regular task, or `TASK_FINAL`, which disallows subsequent `settaskid()` calls by the created task.

The `gettaskid()` function returns the task ID of the calling process.

The `getprojid()` function returns the project ID of the calling process.

Return Values Upon successful completion, these functions return the appropriate task or project ID. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors The `settaskid()` function will fail if:

`EACCES` The invoking task was created with the `TASK_FINAL` flag.

`EAGAIN` A resource control limiting the number of tasks or LWPs in the current project or zone has been exceeded.

A resource control on the given project would be exceeded.

`EINVAL` The given project ID is not within the valid project ID range.

`EPERM` The {`PRIV_PROC_TASKID`} privilege is not asserted in the effective set of the calling process.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

See Also setsid(2), project(4), attributes(5), privileges(5)

Name setuid, setegid, seteuid, setgid – set user and group IDs

Synopsis

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

int setuid(uid_t uid);

int setgid(gid_t gid);

int seteuid(uid_t euid);

int setegid(gid_t egid);
```

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 IDs 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 one of the `exec(2)` family of functions 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.

If the {`PRIV_PROC_SETID`} privilege is asserted in the effective set of the process calling `setuid()`, the real, effective, and saved user IDs are set to the *uid* argument. If the *uid* argument is 0 and none of the saved, effective or real UID is 0, additional restrictions apply. See `privileges(5)`.

If the {`PRIV_PROC_SETID`} privilege is not asserted in the effective set, 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 {`PRIV_PROC_SETID`} privilege is asserted in the effective set of the process calling `setgid()`, the real, effective, and saved group IDs are set to the *gid* argument.

If the {`PRIV_PROC_SETID`} privilege is not asserted in the effective set, 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, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `setuid()` and `setgid()` functions will fail if:

`EINVAL` The value of *uid* or *gid* is out of range.

`EPERM` For `setuid()` and `seteuid()`, the {`PRIV_PROC_SETID`} privilege is not asserted in the effective set of the calling process and the *uid* argument does not match either the real or saved user IDs, or an attempt is made to change to UID 0 and none of the existing UIDs is 0, in which case additional privileges are required.

For `setgid()` and `setegid()`, the {`PRIV_PROC_SETID`} privilege is not asserted in the effective set and the *gid* argument does not match either the real or saved group IDs.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `Intro(2)`, `exec(2)`, `getgroups(2)`, `getuid(2)`, `stat.h(3HEAD)`, `attributes(5)`, `privileges(5)`, `standards(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 The `shmctl()` function 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    /* access permission bits only */
```

This command can be executed only by a process that has appropriate privileges or an effective user ID equal 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. The segment referenced by the identifier will be destroyed when all processes with the segment attached have either detached the segment or exited. If the segment is not attached to any process when `IPC_RMID` is invoked, it will be destroyed immediately. This command can be executed only by a process that has appropriate privileges or an effective user ID equal 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 appropriate privileges.

SHM_UNLOCK Unlock the shared memory segment specified by *shmid*. This command can be executed only by a process that has appropriate privileges.

A shared memory segment must be explicitly removed using IPC_RMID before the system can deallocate it and the resources it uses.

Return Values Upon successful completion, 0 is returned. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The shmctl() function will fail if:

EACCES The *cmd* argument is equal to IPC_STAT and {READ} operation permission is denied to the calling process.

EFAULT The *buf* argument points to an illegal address.

EINVAL The *shmid* argument is not a valid shared memory identifier; or the *cmd* argument is not a valid command or is IPC_SET and shm_perm.uid or shm_perm.gid is not valid.

ENOMEM The *cmd* argument is equal to SHM_LOCK and there is not enough memory, or the operation would exceed a limit or resource control on locked memory.

EOVERFLOW The *cmd* argument is IPC_STAT and *uid* or *gid* is too large to be stored in the structure pointed to by *buf*.

EPERM The *cmd* argument is equal to IPC_RMID or IPC_SET, the effective user ID of the calling process is not equal to the value of shm_perm.cuid or shm_perm.uid in the data structure associated with *shmid*, and {PRIV_IPC_OWNER} is not asserted in the effective set of the calling process.

The *cmd* argument is equal to SHM_LOCK or SHM_UNLOCK and {PRIV_PROC_LOCK_MEMORY} is not asserted in the effective set of the calling process.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also ipcs(1), Intro(2), shmget(2), shmop(2), attributes(5), privileges(5), standards(5)

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, size_t size, int shmflg);
```

Description The `shmget()` function 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:

- The *key* argument is equal to `IPC_PRIVATE`.
- The *key* argument 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:

- The values of `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*.
- The values of `shm_lpid`, `shm_nattch` `shm_atime`, and `shm_dtime` are set equal to 0.
- The `shm_ctime` is set equal to the current time.

Shared memory segments must be explicitly removed after the last reference to them has been removed.

Return Values Upon successful completion, a non-negative integer representing a shared memory identifier is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `shmget()` function will fail if:

`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` The *size* argument is less than the system-imposed minimum or greater than the system-imposed maximum. See NOTES.

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 0.

`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 NOTES.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also rctladm(1M), Intro(2), setrctl(2), shmctl(2), shmop(2), ftok(3C), getpagesize(3C), attributes(5), standards(5)

Notes The project.max-shm-memory resource control restricts the total amount of shared memory a project can allocate. The zone.max-shm-memory resource control restricts the total amount of shared memory that can be allocated by a zone. The system-imposed maximum on the size of a shared memory segment is therefore a function of the sizes of any other shared memory segments the calling project might have allocated that are still in use, as well as any other shared memory segments allocated and still in use by processes in the zone. For accounting purposes, segment sizes are rounded up to the nearest multiple of the system page size. See getpagesize(3C).

The system-imposed limit on the number of shared memory identifiers is maintained on a per-project basis using the project.max-shm-ids resource control. The zone.max-shm-ids resource control restricts the total number of shared memory identifiers that can be allocated by a zone.

See rctladm(1M) and setrctl(2) for information about using resource controls.

Name shmids – discover all shared memory identifiers

Synopsis

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

int shmids(int *buf, uint_t nids, uint_t *pnids);
```

Description The `shmids()` function copies all active shared memory identifiers from the system into the user-defined buffer specified by *buf*, provided that the number of such identifiers is not greater than the number of integers the buffer can contain, as specified by *nids*. If the size of the buffer is insufficient to contain all of the active shared memory identifiers in the system, none are copied.

Whether or not the size of the buffer is sufficient to contain all of them, the number of active shared memory identifiers in the system is copied into the unsigned integer pointed to by *pnids*.

If *nids* is 0 or less than the number of active shared memory identifiers in the system, *buf* is ignored.

Return Values Upon successful completion, `shmids()` returns `0`. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `shmids()` function will fail if:

`EFAULT` The *buf* or *pnids* argument points to an illegal address.

Usage The `shmids()` function returns a snapshot of all the active shared memory identifiers in the system. More may be added and some may be removed before they can be used by the caller.

Examples **EXAMPLE 1** `shmids()` example

This is sample C code indicating how to use the `shmids()` function.

```
void
examine_shmids()
{
     int *ids = NULL;
     uint_t nids = 0;
     uint_t n;
     int i;

     for (;;) {
          if (shmids(ids, nids, &n) != 0) {
               perror("shmids");
               exit(1);
          }
          if (n <= nids)     /* we got them all */
               break;
          /* we need a bigger buffer */
```

EXAMPLE 1 `shmids()` example *(Continued)*

```
            ids = realloc(ids, (nids = n) * sizeof (int));
        }

        for (i = 0; i < n; i++)
            process_shmid(ids[i]);

        free(ids);
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

See Also

`ipcrm(1)`, `ipcs(1)`, `Intro(2)`, `shmctl(2)`, `shmget(2)`, `shmop(2)`, `attributes(5)`

Name shmop, shmat, shmdt – shared memory operations

Synopsis

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

void *shmat(int shmid, const void *shmaddr, int shmflg);

int shmdt(char *shmaddr);
```

Standard conforming

```
int shmdt(const void *shmaddr);
```

Description The `shmat()` function 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.

For shared memory segments created with the `SHM_SHARE_MMU` or `SHM_PAGEABLE` flags, the default protections cannot be changed so as to prevent a single process from affecting other processes sharing the same shared segment.

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.

When (*shmflg*&`SHM_PAGEABLE`) is true, virtual memory resources are shared and the dynamic shared memory (DISM) framework is created. The dynamic shared memory can be resized dynamically within the specified size in `shmget(2)`. The DISM shared memory is pageable unless it is locked.

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`*)* or (*shmflg*&`SHM_PAGEABLE`) is true, then the segment is attached to the first available suitably aligned address. When (*shmflg*&`SHM_SHARE_MMU`) or (*shmflg*&`SHM_PAGEABLE`) is set, however, the permission given by `shmget()` determines whether the segment is attached for reading or reading and writing.
- 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}.

The `shmdt()` function detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr*. If the application is standard-conforming (see `standards(5)`), the *shmaddr* argument is of type `const void *`. Otherwise it is of type `char *`.

Shared memory segments must be explicitly removed after the last reference to them has been removed.

Return Values Upon successful completion, `shmat()` returns the data segment start address of the attached shared memory segment; `shmdt()` returns `0`. Otherwise, `−1` is returned, the shared memory segment is not attached, and `errno` is set to indicate the error.

Errors The `shmat()` function will fail if:

`EACCES` Operation permission is denied to the calling process (see `Intro(2)`).

`EINVAL` The *shmid* argument is not a valid shared memory identifier.

The *shmaddr* argument is not equal to 0, and the value of (*shmaddr* - (*shmaddr* modulus `SHMLBA`)) is an illegal address.

The *shmaddr* argument is not equal to 0, is an illegal address, and (*shmflg*&`SHM_RND`) is false.

The *shmaddr* argument is not equal to 0, is not properly aligned, and (*shmfg*&`SHM_SHARE_MMU`) is true.

`SHM_SHARE_MMU` is not supported in certain architectures.

Both (*shmflg*&`SHM_SHARE_MMU`) and *(shmflg*&`SHM_PAGEABLE`) are true.

(*shmflg*&`SHM_SHARE_MMU`) is true and the shared memory segment specified by `shmid()` had previously been attached by a call to `shmat()` in which (*shmflg*&`SHM_PAGEABLE`) was true.

(*shmflg*&`SHM_PAGEABLE`) is true and the shared memory segment specified by `shmid()` had previously been attached by a call to `shmat()` in which (*shmflg*&`SHM_SHARE_MMU`) was true.

`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.

The `shmdt()` function will fail if:

`EINVAL` The *shmaddr* argument is not the data segment start address of a shared memory segment.

`ENOMEM` (*shmflg*&`SHM_SHARE_MMU`) is true and attaching to the shared memory segment would exceed a limit or resource control on locked memory.

Warnings Using a fixed value for the *shmaddr* argument can adversely affect performance on certain platforms due to D-cache aliasing.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See `standards(5)`.

See Also `Intro(2)`, `exec(2)`, `exit(2)`, `fork(2)`, `shmctl(2)`, `shmget(2)`, `attributes(5)`, `standards(5)`

Name sigaction – detailed signal management

Synopsis

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *restrict act,
     struct sigaction *restrict oact);
```

Description The `sigaction()` function allows the calling process to examine or specify the action to be taken on delivery of a specific signal. See `signal.h`(3HEAD) for an explanation of general signal concepts.

The *sig* argument specifies the signal and can be assigned any of the signals specified in `signal.h`(3HEAD) except `SIGKILL` and `SIGSTOP`.

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;
```

The storage occupied by `sa_handler` and `sa_sigaction` may overlap, and a standard-conforming application (see `standards`(5)) must not use both simultaneously.

The `sa_handler` member 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.h`(3HEAD) 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.

The `sa_mask` member 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).

The `sa_flags` member 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 thread 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 thread's normal stack.

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, functions that are interrupted by the execution of this signal's handler are transparently restarted by the system, namely fcntl(2), ioctl(2), wait(3C), waitid(2), and the following functions on slow devices like terminals: getmsg() and getpmsg() (see getmsg(2)); putmsg() and putpmsg() (see putmsg(2)); pread(), read(), and readv() (see read(2)); pwrite(), write(), and writev() (see write(2)); recv(), recvfrom(), and recvmsg() (see recv(3SOCKET)); and send(), sendto(), and sendmsg() (see send(3SOCKET)). Otherwise, the 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, 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.h(3HEAD)); the third argument points to a ucontext_t structure containing the receiving process's context when the signal was delivered (see ucontext.h(3HEAD)).

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(3C), it blocks until all of the calling process's child processes terminate, and then returns −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.

Return Values Upon successful completion, 0 is returned. Otherwise, −1 is returned, errno is set to indicate the error, and no new signal handler is installed.

Errors The sigaction() function will fail if:

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 multithreaded process, it is equal to SIGWAITING, SIGCANCEL, or SIGLWP.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See standards(5).

See Also kill(1), Intro(2), exit(2), fcntl(2), getmsg(2), ioctl(2), kill(2), pause(2), putmsg(2), read(2), sigaltstack(2), sigprocmask(2), sigsend(2), sigsuspend(2), waitid(2), write(2), recv(3SOCKET), send(3SOCKET), siginfo.h(3HEAD), signal(3C), signal.h(3HEAD), sigsetops(3C), ucontext.h(3HEAD), wait(3C), attributes(5), standards(5)

Notes The handler routine can be declared:

```
void handler (int sig, siginfo_t *sip, ucontext_t *ucp);
```

The *sig* argument is the signal number. The *sip* argument is a pointer (to space on the stack) to a `siginfo_t` structure, which provides additional detail about the delivery of the signal. The *ucp* argument 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 *ucp* 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 *restrict ss, stack_t *restrict oss);
```

Description The `sigaltstack()` function allows a thread to define and examine the state of an alternate stack area on which signals are processed. If *ss* is non-zero, it specifies a pointer to and the size of a stack area on which to deliver signals, and informs the system whether the thread 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 whether the thread chosen to execute the signal handler is currently executing on that stack. If the thread 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 `stack_t` structure 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` members specify the new base and size of the stack, which is automatically adjusted for direction of growth and alignment. The `ss_flags` member 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` members specify the base and size of that stack. The `ss_flags` member specifies the stack's state, and may contain the following values:

`SS_ONSTACK` The thread is currently executing on the alternate signal stack. Attempts to modify the alternate signal stack while the thread is executing on it will fail.

`SS_DISABLE` The alternate signal stack is currently disabled.

Return Values Upon successful completion, `0` is return. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `sigaltstack()` function will fail if:

`EFAULT` The *ss* or *oss* argument points to an illegal address.

`EINVAL` The *ss* argument is not a null pointer, and the `ss_flags` member pointed to by *ss* contains flags other than `SS_DISABLE`.

ENOMEM The size of the alternate stack area is less than MINSIGSTKSZ.

EPERM An attempt was made to modify an active stack.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also getcontext(2), mmap(2), sigaction(2), ucontext.h(3HEAD), attributes(5), standards(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 with an adjacent red zone (an unmapped page) to guard against stack overflow, as with default stacks:

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

stack_t sigstk;
sigstk.ss_sp = mmap(NULL, SIGSTKSZ, PROT_READ | PROT_WRITE,
        MAP_PRIVATE | MAP_ANON, -1, 0);
if (sigstk.ss_sp == MAP_FAILED)
        /* error return */;
sigstk.ss_size = SIGSTKSZ;
sigstk.ss_flags = 0;
if (sigaltstack(&sigstk, NULL) < 0)
        perror("sigaltstack");
```

Name sigpending – examine signals that are blocked and pending

Synopsis

```
#include <signal.h>

int sigpending(sigset_t *set);
```

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 *set* argument.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `sigpending()` function will fail if:

`EFAULT` The *set* argument points to an illegal address.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `sigaction(2)`, `sigprocmask(2)`, `sigsetops(3C)`, `attributes(5)`, `standards(5)`

Name sigprocmask – change or examine caller's signal mask

Synopsis

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *restrict set,
     sigset_t *restrict oset);
```

Description The `sigprocmask()` function is used to examine and/or change the caller's signal mask. If the value is `SIG_BLOCK`, the set pointed to by the *set* argument is added to the current signal mask. If the value is `SIG_UNBLOCK`, the set pointed by the *set* argument 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 *set* argument. If the *oset* argument is not `NULL`, the previous mask is stored in the space pointed to by *oset*. If the value of the *set* argument is `NULL`, the value *how* is not significant and the caller's signal mask is unchanged; thus, the call can be used to inquire about currently blocked signals. If the *set* or *oset* argument points to an invalid address, the behavior is undefined and `errno` may be set to `EFAULT`.

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 signals that cannot be caught or ignored (see `sigaction(2)`). It is also not possible to block or unblock SIGCANCEL, as SIGCANCEL is reserved for the implementation of POSIX thread cancellation (see `pthread_cancel(3C)` and `cancellation(5)`). This restriction is silently enforced by the standard C library.

If `sigprocmask()` fails, the caller's signal mask is not changed.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `sigprocmask()` function will fail if:

`EINVAL` The value of the *how* argument is not equal to one of the defined values.

The `sigprocmask()` function may fail if:

`EFAULT` The *set* or *oset* argument points to an illegal address.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `sigaction(2)`, `pthread_cancel(3C)`, `pthread_sigmask(3C)`, `signal(3C)`, `signal.h(3HEAD)`, `sigsetops(3C)`, `attributes(5)`, `cancellation(5)`

Notes The call to `sigprocmask()` affects only the calling thread's signal mask. It is identical to a call to `pthread_sigmask(3C)`.

Signals that are generated synchronously should not be masked. If such a signal is blocked and delivered, the receiving process is killed.

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 The `sigsend()` function 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 0 or one of the values listed in `signal.h(3HEAD)`. If *sig* is 0 (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 saved user ID of the receiving process, unless the {`PRIV_PROC_OWNER`} privilege is asserted in the effective set of the sending process 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 all processes with process group ID *id*.

If *idtype* is `P_SID`, *sig* is sent to all processes with session ID *id*.

If *idtype* is `P_TASKID`, *sig* is sent to all processes with task 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_PROJID`, *sig* is sent to any process with project 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_CTID`, *sig* is sent to any process with process contract ID *id*.

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`.

The `sigsendset()` function 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;
```

The `p_lidtype` and `p_lid` members specify the ID type and ID of one ("left") set of processes; the `p_ridtype` and `p_rid` members 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()`. The `p_op` member 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 Upon successful completion, `0` is return. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `sigsend()` and `sigsendset()` functions will fail if:

`EINVAL` The *sig* argument is not a valid signal number, or the *idtype* argument is not a valid idtype field.

`EINVAL` The *sig* argument is `SIGKILL`, *idtype* is `P_PID` and *id* is `1` (proc1).

`EPERM` The effective user of the calling process does not match the real or saved user ID of the receiving process, the calling process does not have the {`PRIV_PROC_OWNER`} privilege asserted in the effective set, and the calling process is not sending `SIGCONT` to a process that shares the same session ID.

The calling process does not have the {`PRIV_PROC_SESSION`} privilege asserted and is trying to send a signal to a process with a different session ID, even though the effective user ID matches the real or saved ID of the receiving process.

`ESRCH` No process can be found corresponding to that specified by *id* and *idtype*.

The `sigsendset()` function will fail if:

`EFAULT` The *psp* argument points to an illegal address.

See Also `kill(1)`, `getpid(2)`, `kill(2)`, `priocntl(2)`, `signal(3C)`, `signal.h(3HEAD)`, `process(4)`, `privileges(5)`

Name sigsuspend – install a signal mask and suspend caller until signal

Synopsis

```
#include <signal.h>

int sigsuspend(const sigset_t *set);
```

Description The `sigsuspend()` function replaces the caller's signal mask with the set of signals pointed to by the *set* argument and suspends the caller until delivery of a signal whose action is either to execute a signal catching function or to terminate the process. If the *set* argument points to an invalid address, the behavior is undefined and `errno` may be set to `EFAULT`.

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 signals that cannot be ignored (see `signal.h(3HEAD)`). This restriction is silently imposed by the system.

Return Values Since `sigsuspend()` suspends the caller's execution indefinitely, there is no successful completion return value. On failure, it returns `–1` and sets `errno` to indicate the error.

Errors The `sigsuspend()` function will fail if:

`EINTR` A signal was caught by the caller and control was returned from the signal catching function.

The `sigsuspend()` function may fail if:

`EFAULT` The *set* argument points to an illegal address.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `sigaction(2)`, `sigprocmask(2)`, `sigwait(2)`, `signal(3C)`, `signal.h(3HEAD)`, `sigsetops(3C)`, `attributes(5)`

Notes If the caller specifies more than one unblocked signal in the mask to `sigsuspend()`, more than one signal might be processed before the call to `sigsuspend()` returns.

While the caller is executing the signal handler that interrupted its call to `sigsuspend()`, its signal mask is the one passed to `sigsuspend()`, modified as usual by the signal mask

specification in the signal's `sigaction(2)` parameters. The caller's signal mask is not restored to its previous value until the caller returns from all the signal handlers that interrupted `sigsuspend()`.

Name sigwait – wait until a signal is posted

Synopsis

```
#include <signal.h>

int sigwait(sigset_t *set);
```

Standard conforming

```
cc [ flag ... ] file ... -D_POSIX_PTHREAD_SEMANTICS [ library...]
#include <signal.h>

int sigwait(const sigset_t *set, int *sig);
```

Description The `sigwait()` function selects a signal in *set* that is pending on the calling thread. If no signal in *set* is pending, `sigwait()` blocks until a signal in *set* becomes pending. The selected signal is cleared from the set of signals pending on the calling thread and the number of the signal is returned, or in the standard–conforming version (see `standards(5)`) placed in *sig*. The selection of a signal in *set* is independent of the signal mask of the calling thread. This means a thread can synchronously wait for signals that are being blocked by the signal mask of the calling thread . To ensure that only the caller receives the signals defined in *set*, all threads should have signals in *set* masked including the calling thread.

If more than one thread is using `sigwait()` to wait for the same signal, no more than one of these threads returns from `sigwait()` with the signal number. If more than a single thread is blocked in `sigwait()` for a signal when that signal is generated for the process, it is unspecified which of the waiting threads returns from `sigwait()`. If the signal is generated for a specific thread, as by `pthread_kill(3C)`, only that thread returns.

Should any of the multiple pending signals in the range `SIGRTMIN` to `SIGRTMAX` be selected, it will be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified.

Return Values Upon successful completion, the default version of `sigwait()` returns a signal number; the standard–conforming version returns `0` and stores the received signal number at the location pointed to by *sig*. Otherwise, the default version returns -1 and sets errno to indicate an error; the standard-conforming version returns an error number to indicate the error.

Errors The `sigwait()` function will fail if:

`EFAULT` The *set* argument points to an invalid address.

`EINTR` The wait was interrupted by an unblocked, caught signal.

`EINVAL` The *set* argument contains an unsupported signal number.

Examples **EXAMPLE 1** Creating a thread to handle receipt of a signal

The following sample C code creates a thread to handle the receipt of a signal. More specifically, it catches the asynchronously generated signal, `SIGINT`.

```
/*****************************************************************
*
```

EXAMPLE 1 Creating a thread to handle receipt of a signal *(Continued)*

```
* compile with -D_POSIX_PTHREAD_SEMANTICS switch;
* required by sigwait()
*
* sigint thread handles delivery of signal. uses sigwait( ) to wait
* for SIGINT signal.
*
*****************************************************************/
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <synch.h>

static void    *threadTwo(void *);
static void    *threadThree(void *);
static void    *sigint(void *);

sigset_t       signalSet;

void *
main(void)
{
    pthread_t    t;
    pthread_t    t2;
    pthread_t    t3;

    sigfillset ( &signalSet );
    /*
     * Block signals in initial thread. New threads will
     * inherit this signal mask.
     */
    pthread_sigmask ( SIG_BLOCK, &signalSet, NULL );

    printf("Creating threads\n");

    pthread_create(&t, NULL, sigint, NULL);
    pthread_create(&t2, NULL, threadTwo, NULL);
    pthread_create(&t3, NULL, threadThree, NULL);

    printf("##################\n");
    printf("press CTRL-C to deliver SIGINT to sigint thread\n");
    printf("##################\n");
```

EXAMPLE 1 Creating a thread to handle receipt of a signal *(Continued)*

```
    pthread_exit((void *)0);
}
static void *
threadTwo(void *arg)
{
    printf("hello world, from threadTwo [tid: %d]\n",
                            pthread_self());
    printf("threadTwo [tid: %d] is now complete and exiting\n",
                            pthread_self());
    pthread_exit((void *)0);
}

static void *
threadThree(void *arg)
{
    printf("hello world, from threadThree [tid: %d]\n",
                            pthread_self());
    printf("threadThree [tid: %d] is now complete and exiting\n",
                            pthread_self());
    pthread_exit((void *)0);
}

void *
sigint(void *arg)
{
    int    sig;
    int    err;

    printf("thread sigint [tid: %d] awaiting SIGINT\n",
                            pthread_self());

    /*
    /* use standard-conforming sigwait() -- 2 args: signal set, signum
     */
    err = sigwait ( &signalSet, &sig );

    /* test for SIGINT; could catch other signals */
    if (err || sig != SIGINT)
        abort();

    printf("\nSIGINT signal %d caught by sigint thread [tid: %d]\n",
                            sig, pthread_self());
    pthread_exit((void *)0);
}
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See standards(5).

See Also sigaction(2), sigpending(2), sigprocmask(2), sigsuspend(2), pthread_create(3C), pthread_kill(3C), pthread_sigmask(3C), signal.h(3HEAD), attributes(5), standards(5)

Notes The sigwait() function cannot be used to wait for signals that cannot be caught (see sigaction(2)). This restriction is silently imposed by the system.

Solaris 2.4 and earlier releases provided a sigwait() facility as specified in POSIX.1c Draft 6. The final POSIX.1c standard changed the interface as described above. Support for the Draft 6 interface is provided for compatibility only and may not be supported in future releases. New applications and libraries should use the standard–conforming interface.

Name __sparc_utrap_install – install a SPARC V9 user trap handler

Synopsis

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

int __sparc_utrap_install(utrap_entry_t type,
     utrap_handler_t new_precise, utrap_handler_t new_deferred,
     utrap_handler_t *old_precise, utrap_handler_t *old_deferred);
```

Description The `__sparc_utrap_install()` function establishes *new_precise* and *new_deferred* user trap handlers as the new values for the specified *type* and returns the existing user trap handler values in **old_precise* and **old_deferred* in a single atomic operation. A new handler address of `NULL` means no user handler of that type will be installed. A new handler address of `UTH_NOCHANGE` means that the user handler for that type should not be changed. An old handler pointer of `NULL` means that the user is not interested in the old handler address.

A *precise trap* is caused by a specific instruction and occurs before any program-visible state has been changed by this instruction. When a precise trap occurs, the program counter (PC) saved in the Trap Program Counter (TPC) register points to the instruction that induced the trap; all instructions prior to this trapping instruction have been executed. The next program counter (nPC) saved in the Trap Next Program Counter (TnPC) register points to the next instruction following the trapping instruction, which has not yet been executed. A *deferred trap* is also caused by a particular instruction, but unlike a precise trap, a deferred trap may occur after the program-visible state has been changed. See the *SPARC Architecture Manual, Version 9* for further information on precise and deferred traps.

The list that follows contains hardware traps and their corresponding user trap types. User trap types marked with a plus-sign (+) are required and must be provided by all ABI-conforming implementations. The others may not be present on every implementation; an attempt to install a user trap handler for those conditions will return `EINVAL`. User trap types marked with an asterisk (*) are implemented as precise traps only.

Trap Name	User Trap Type (utrap_entry_t)
illegal_instruction	UT_ILLTRAP_INSTRUCTION +* or UT_ILLEGAL_INSTRUCTION
fp_disabled	UT_FP_DISABLED +*
fp_exception_ieee_754	UT_FP_EXCEPTION_IEEE_754 +
fp_exception_other	UT_FP_EXCEPTION_OTHER
tag_overflow	UT_TAG_OVERFLOW +*
division_by_zero	UT_DIVISION_BY_ZERO +
mem_address_not_aligned	UT_MEM_ADDRESS_NOT_ALIGNED +

Trap Name	User Trap Type (utrap_entry_t)
privileged_action	UT_PRIVILEGED_ACTION +
privileged_opcode	UT_PRIVILEGED_OPCODE
async_data_error	UT_ASYNC_DATA_ERROR
trap_instruction	UT_TRAP_INSTRUCTION_16 through UT_TRAP_INSTRUCTION_31 +*
instruction_access_exception instruction_access_MMU_miss instruction_access_error	UT_INSTRUCTION_EXCEPTION or UT_INSTRUCTION_PROTECTION or UT_INSTRUCTION_ERROR
data_access_exception data_access_MMU_miss data_access_error data_access_protection	UT_DATA_EXCEPTION or UT_DATA_PROTECTION or UT_DATA_ERROR

The following explanations are provided for those user trap types that are not self-explanatory.

UT_ILLTRAP_INSTRUCTION
: This trap is raised by user execution of the ILLTRAP INSTRUCTION. It is always precise.

UT_ILLEGAL_INSTRUCTION
: This trap will be raised by the execution of otherwise undefined opcodes. It is implementation-dependent as to what opcodes raise this trap; the ABI only specifies the interface. The trap may be precise or deferred.

UT_PRIVILEGED_OPCODE
: All opcodes declared to be privileged in SPARC V9 will raise this trap. It is implementation-dependent whether other opcodes will raise it as well; the ABI only specifies the interface.

UT_DATA_EXCEPTION, UT_INSTRUCTION_EXCEPTION
: No valid user mapping can be made to this address, for a data or instruction access, respectively.

UT_DATA_PROTECTION, UT_INSTRUCTION_PROTECTION
: A valid mapping exists, and user privilege to it exists, but the type of access (read, write, or execute) is denied, for a data or instruction access, respectively.

UT_DATA_ERROR, UT_INSTRUCTION_ERROR
: A valid mapping exists, and both user privilege and the type of access are allowed, but an unrecoverable error occurred in attempting the access, for a data or instruction access, respectively. %l1 will contain either BUS_ADDRERR or BUS_OBJERR.

UT_FP_DISABLED
: This trap is raised when an application issues a floating point instruction (including load or store) and the SPARC V9 Floating Point Registers State (FPRS) FEF bit is 0. If a user handler is installed for this trap, it will be given control. Otherwise the system will set FEF

to one and retry the instruction.

For all traps, the handler executes in a new register window, where the *in* registers are the *out* registers of the previous frame and have the value they contained at the time of the trap, similar to a normal subroutine call after the `save` instruction. The *global* registers (including the special registers `%ccr`, `%asi`, and `%y`) and the *floating-point* registers have their values from the time of the trap. The stack pointer register `%sp` plus the BIAS will point to a properly-aligned 128-byte register save area; if the handler needs scratch space, it should decrement the stack pointer to obtain it. If the handler needs access to the previous frame's *in* registers or *local* registers, it should execute a `FLUSHW` instruction, and then access them off of the frame pointer. If the handler calls an ABI-conforming function, it must set the `%asi` register to `ASI_PRIMARY_NOFAULT` before the call.

On entry to a precise user trap handler `%l6` contains the `%pc` and `%l7` contains the `%npc` at the time of the trap. To return from a handler and reexecute the trapped instruction, the handler would execute:

```
jmpl %l6, %g0 ! Trapped PC supplied to user trap handler
return %l7    ! Trapped nPC supplied to user trap handler
```

To return from a handler and skip the trapped instruction, the handler would execute:

```
jmpl %l7, %g0 ! Trapped nPC supplied to user trap handler
return %l7 + 4 ! Trapped nPC + 4
```

On entry to a deferred trap handler `%o0` contains the address of the instruction that caused the trap and `%o1` contains the actual instruction (right-justified, zero-extended), if the information is available. Otherwise `%o0` contains the value −1 and `%o1` is undefined. Additional information may be made available for certain cases of deferred traps, as indicated in the following table.

Instructions	Additional Information
LD-type (LDSTUB)	`%o2` contains the effective address (*rs1* + *rs2* | *simm13*).
ST-type (CAS, SWAP)	`%o2` contains the effective address (*rs1* + *rs2* |*simm13*).
Integer arithmetic	`%o2` contains the *rs1* value. `%o3` contains the *rs2* | *simm13* value. `%o4` contains the contents of the `%y` register.
Floating-point arithmetic	`%o2` contains the address of *rs1* value. `%o3` contains the address of *rs2* value.
Control-transfer	`%o2` contains the target address (*rs1* + *rs2* | *simm13*).
Asynchronous data errors	`%o2` contains the address that caused the error. `%o3` contains the effective ASI, if available, else −1.

To return from a deferred trap, the trap handler issues:

ta 68 ! ST_RETURN_FROM_DEFERRED_TRAP

The following pseudo-code explains how the operating system dispatches traps:

```
if (precise trap) {
      if (precise_handler) {
           invoke(precise_handler);
           /* not reached */
      } else {
           convert_to_signal(precise_trap);
      }
 } else if (deferred_trap) {
      invoke(deferred_handler);
           /* not reached */
      } else {
           convert_to_signal(deferred_trap);
      }
 }
 if (signal)
           send(signal);
```

User trap handlers must preserve all registers except the *locals* (`%l0-7`) and the *outs* (`%o0-7`), that is, `%i0-7`, `%g1-7`, `%d0-d62`, `%asi`, `%fsr`, `%fprs`, `%ccr`, and `%y`, except to the extent that modifying the registers is part of the desired functionality of the handler. For example, the handler for `UT_FP_DISABLED` may load floating-point registers.

Return Values Upon successful completion, `0` is returned. Otherwise, a non-zero value is returned and `errno` is set to indicate the error.

Errors The `__sparc_utrap_install()` function will fail if:

`EINVAL` The *type* argument is not a supported user trap type; the new user trap handler address is not word aligned; the old user trap handler address cannot be returned; or the user program is not a 64-bit executable.

Examples **EXAMPLE 1** A sample program using the `__sparc_utrap_install()` function.

The `__sparc_utrap_install()` function is normally used by user programs that wish to provide their own tailored exception handlers as a faster alternative to `signal(3C)`, or to handle exceptions that are not directly supported by the `signal()` interface, such as `fp_disabled`.

```
extern void *fpdis_trap_handler();
utrap_handler_t new_precise = (utrap_handler_t)fpdis_trap_handler;
double d;
int err;
err = __sparc_utrap_install(UT_FP_DISABLED, new_precise,
    UTH_NOCHANGE, NULL, NULL);
if (err == EINVAL) {
        /* unexpected error, do something */
        exit (1);
```

EXAMPLE 1 A sample program using the `__sparc_utrap_install()` function. *(Continued)*

```
}
d = 1.0e-300;
ENTRY(fpdis_trap_handler)
wr      %g0, FPRS_FEF, %fprs
jmpl    %l6, %g0
return  %l7
SET_SIZE(fpdis_trap_handler)
```

This example turns on bit 2, FEF, in the Floating-Point Registers State (FPRS) Register, after a floating-point instruction causes an `fp_disabled` trap. (Note that this example simulates part of the default system behavior; programs do not need such a handler. The example is for illustrative purposes only.)

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

See Also

`signal(3C)`, `attributes(5)`

SPARC Architecture Manual, Version 9

Manufacturer's processor chip user manuals

Notes

The Exceptions and Interrupt Descriptions section of the SPARC V9 manual documents which hardware traps are mandatory or optional, and whether they can be implemented as precise or deferred traps, or both. The manufacturer's processor chip user manuals describe the details of the traps supported for the specific processor implementation.

Name stat, lstat, fstat, fstatat – get file status

Synopsis

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

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

int lstat(const char *restrict path, struct stat *restrict buf);

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

int fstatat(int fildes, const char *path, struct stat *buf,
     int flag);
```

Description The `stat()` function 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.

The `lstat()` function 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.

The `fstat()` function obtains information about an open file known by the file descriptor *fildes*, obtained from a successful `open(2)`, `creat(2)`, `dup(2)`, `fcntl(2)`, or `pipe(2)` function. If *fildes* references a shared memory object, the system updates in the `stat` structure pointed to by the *buf* argument only the `st_uid`, `st_gid`, `st_size`, and `st_mode` fields, and only the `S_IRUSR`, `S_IWUSR`, `S_IRGRP`, `S_IWGRP`, `S_IROTH`, and `S_IWOTH` file permission bits need be valid. The system can update other fields and flags. The `fstat()` function updates any pending time-related fields before writing to the `stat` structure.

The `fstatat()` function obtains file attributes similar to the `stat()`, `lstat()`, and `fstat()` functions. If the *path* argument is a relative path, it is resolved relative to the *fildes* argument rather than the current working directory. If *path* is absolute, the *fildes* argument is unused. If the *fildes* argument has the special value `AT_FDCWD`, relative paths are resolved from the current working directory. If `AT_SYMLINK_NOFOLLOW` is set in the *flag* argument, the function behaves like `lstat()` and does not automatically follow symbolic links. See `fsattr(5)`. If `_ATTR_TRIGGER` is set in the *flag* argument and the vnode is a trigger mount point, the mount is performed and the function returns the attributes of the root of the mounted filesystem.

The *buf* argument is a pointer to a `stat` structure into which information is placed concerning the file. A `stat` structure includes 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 */
blkcnt_t st_blocks;       /* Number of 512 byte blocks allocated*/
char     st_fstype[_ST_FSTYPSZ];
                          /* Null-terminated type of filesystem */
```

Descriptions of structure members are as follows:

`st_mode` The mode of the file as described for the `mknod()` function. In addition to the modes described on the mknod(2) manual page, the mode of a file can also be `S_IFSOCK` if the file is a socket, `S_IFDOOR` if the file is a door, `S_IFPORT` if the file is an event port, or `S_IFLNK` if the file is a symbolic link. `S_IFLNK` can be returned either by `lstat()` or by `fstat()` when the `AT_SYMLINK_NOFOLLOW` flag is set.

`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. Some of the functions that change this member are: `creat()`, `mknod()`, `pipe()`, `utime(2)`, and `read(2)`.

`st_mtime` Time when data was last modified. Some of the functions that change this member are: `creat()`, `mknod()`, `pipe()`, `utime()`, and `write(2)`.

`st_ctime` Time when file status was last changed. Some of the functions that change this member are: `chmod(2)`, `chown(2)`, `creat(2)`, `link(2)`, `mknod(2)`, `pipe(2)`, `rename(2)`, `unlink(2)`, `utime(2)`, and `write(2)`.

`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.

`st_fstype` A null-teminated string that uniquely identifies the type of the filesystem that contains the file.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `stat()`, `fstat()`, `lstat()`, and `fstatat()` functions will fail if:

`EIO` An error occurred while reading from the file system.

`EOVERFLOW` The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by *buf*.

The `stat()`, `lstat()`, and `fstatat()` functions will fail if:

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

`EFAULT` The *buf* or *path* argument points to an illegal address.

`EINTR` A signal was caught during the execution of the `stat()` or `lstat()` function.

`ELOOP` A loop exists in symbolic links encountered during the resolution of the *path* argument.

`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 *path* does not name an existing file or *path* is an empty string.

`ENOLINK` The *path* argument 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, or the *fildes* argument does not refer to a valid directory when given a non-null relative path.

The `fstat()` and `fstatat()` functions will fail if:

EBADF The *fildes* argument is not a valid open file descriptor. The *fildes* argument to `fstatat()` can also have the valid value of `AT_FDCWD`.

EFAULT The *buf* argument points to an illegal address.

EINTR A signal was caught during the execution of the `fstat()` function.

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

The `stat()`, `fstat()`, and `lstat()` functions may fail if:

EOVERFLOW One of the members is too large to store in the `stat` structure pointed to by *buf*.

The `stat()` and `lstat()` functions may fail if:

ELOOP More than {`SYMLOOP_MAX`} symbolic links were encountered during the resolution of the *path* argument.

ENAMETOOLONG As a result of encountering a symbolic link in resolution of the*path* argument, the length of the substituted pathname strings exceeds {`PATH_MAX`}.

The `stat()` and `fstatat()` functions may fail if:

ENXIO The *path* argument names a character or block device special file and the corresponding I/O device has been retired by the fault management framework.

Examples

EXAMPLE 1 Use `stat()` to obtain file status information.

The following example shows how to obtain file status information for a file named `/home/cnd/mod1`. The structure variable buffer is defined for the `stat` structure.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
struct stat buffer;
int         status;
...
status = stat("/home/cnd/mod1", &buffer);
```

EXAMPLE 2 Use `stat()` to get directory information.

The following example fragment gets status information for each entry in a directory. The call to the `stat()` function stores file information in the `stat` structure pointed to by *statbuf*. The lines that follow the `stat()` call format the fields in the `stat` structure for presentation to the user of the program.

EXAMPLE 2 Use `stat()` to get directory information. *(Continued)*

```
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>
#include <locale.h>
#include <langinfo.h>
#include <stdio.h>
#include <stdint.h>
struct dirent *dp;
struct stat   statbuf;
struct passwd *pwd;
struct group  *grp;
struct tm     *tm;
char          datestring[256];
...
/* Loop through directory entries */
while ((dp = readdir(dir)) != NULL) {
    /* Get entry's information. */
    if (stat(dp->d_name, &statbuf) == -1)
    continue;

     /* Print out type, permissions, and number of links. */
     printf("%10.10s", sperm (statbuf.st_mode));
     printf("%4d", statbuf.st_nlink);

     /* Print out owners name if it is found using getpwuid(). */
     if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
        printf(" %-8.8s", pwd->pw_name);
     else
        printf(" %-8d", statbuf.st_uid);

     /* Print out group name if it's found using getgrgid(). */
     if ((grp = getgrgid(statbuf.st_gid)) != NULL)
        printf(" %-8.8s", grp->gr_name);
     else
        printf(" %-8d", statbuf.st_gid);

     /* Print size of file. */
     printf(" %9jd", (intmax_t)statbuf.st_size);
     tm = localtime(&statbuf.st_mtime);

     /* Get localized date string. */
     strftime(datestring, sizeof(datestring), nl_langinfo(D_T_FMT), tm);
```

EXAMPLE 2 Use `stat()` to get directory information. *(Continued)*

```
    printf(" %s %s\n", datestring, dp->d_name);
}
```

EXAMPLE 3 Use `fstat()` to obtain file status information.

The following example shows how to obtain file status information for a file named `/home/cnd/mod1`. The structure variable buffer is defined for the `stat` structure. The `/home/cnd/mod1` `file` is opened with read/write privileges and is passed to the open file descriptor *fildes*.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
struct stat buffer;
int         status;
...
fildes = open("/home/cnd/mod1", O_RDWR);
status = fstat(fildes, &buffer);
```

EXAMPLE 4 Use `lstat()` to obtain symbolic link status information.

The following example shows how to obtain status information for a symbolic link named `/modules/pass1`. The structure variable buffer is defined for the `stat` structure. If the *path* argument specified the filename for the file pointed to by the symbolic link (`/home/cnd/mod1`), the results of calling the function would be the same as those returned by a call to the `stat()` function.

```
#include <sys/stat.h>
struct stat buffer;
int         status;
...
status = lstat("/modules/pass1", &buffer);
```

Usage If `chmod()` or `fchmod()` is used to change the file group owner permissions on a file with non-trivial ACL entries, only the ACL mask is set to the new permissions and the group owner permission bits in the file's mode field (defined in `mknod(2)`) are unchanged. A non-trivial ACL entry is one whose meaning cannot be represented in the file's mode field alone. The new ACL mask permissions might change the effective permissions for additional users and groups that have ACL entries on the file.

The `stat()`, `fstat()`, and `lstat()` functions have transitional interfaces for 64-bit file offsets. See `lf64(5)`.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See below.

For stat(), fstat(), and lstat(), see standards(5).

See Also access(2), 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.h(3HEAD), attributes(5), fsattr(5), lf64(5), standards(5)

Name statvfs, fstatvfs – get file system information

Synopsis

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

int statvfs(const char *restrict path, struct statvfs *restrict buf);

int fstatvfs(int fildes, struct statvfs *buf);
```

Description The `statvfs()` function returns a "generic superblock" describing a file system; it can be used to acquire information about mounted file systems. The *buf* argument is a pointer to a structure (described below) that is filled by the function.

The *path* argument 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) */
fsblkcnt_t  f_blocks;            /* total # of blocks on file system
                                    in units of f_frsize */
fsblkcnt_t  f_bfree;             /* total # of free blocks */
fsblkcnt_t  f_bavail;            /* # of free blocks avail to
                                    non-privileged user */
fsfilcnt_t  f_files;             /* total # of file nodes (inodes) */
fsfilcnt_t  f_ffree;             /* total # of free file nodes */
fsfilcnt_t  f_favail;            /* # of inodes avail to
                                    non-privileged 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 */
```

The `f_basetype` member contains a null-terminated FSType name of the mounted target.

The following values 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 */
```

The `fstatvfs()` function 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 call.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `statvfs()` and `fstatvfs()` functions will fail if:

`EOVERFLOW` One of the values to be returned cannot be represented correctly in the structure pointed to by *buf*.

The `statvfs()` function will fail if:

`EACCES` Search permission is denied on a component of the path prefix.

`EFAULT` The *path* or *buf* argument points to an illegal address.

`EINTR` A signal was caught during the execution of the `statvfs()` function.

`EIO` An I/O error occurred while reading the file system.

`ELOOP` Too many symbolic links were encountered in translating *path*.

`ENAMETOOLONG` The length of a *path* component exceeds `NAME_MAX` characters, or the length of *path* The exceeds `PATH_MAX` characters.

`ENOENT` Either a component of the path prefix or the file referred to by *path* does not exist.

`ENOLINK` The *path* argument 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.

The `fstatvfs()` function will fail if:

`EBADF` The *fildes* argument is not an open file descriptor.

`EFAULT` The *buf* argument points to an illegal address.

`EINTR` A signal was caught during the execution of the `fstatvfs()` function.

`EIO` An I/O error occurred while reading the file system.

Usage The `statvfs()` and `fstatvfs()` functions have transitional interfaces for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

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), attributes(5), lf64(5), standards(5)

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 The `stime()` function sets the system's idea of the time and date. The *tp* argument points to the value of time as measured in seconds from 00:00:00 UTC January 1, 1970.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `stime()` function will fail if:

`EINVAL` The *tp* argument points to an invalid (negative) value.

`EPERM` The {`PRIV_SYS_TIME`} privilege is not asserted in the effective set of the calling process.

See Also `time(2)`, `privileges(5)`

Name swapctl – manage swap space

Synopsis

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

int swapctl(int cmd, void *arg);
```

Description The `swapctl()` function 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 */
```

The `sr_start` and `sr_length` members are specified in 512-byte blocks. A swap resource can only be removed by specifying the same values for the `sr_start` and `sr_length` members as were specified when it was added. Swap resources need not be removed in the order in which they were added.

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 */
```

The `SC_LIST` function 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	Either *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 {PRIV_SYS_MOUNT} was not asserted in the effective set of the calling process.
EROFS	The pathname specified for SC_ADD is a read-only file system.

Additionally, the swapctl() function will fail for 32-bit interfaces if:

EOVERFLOW	The amount of swap space configured on the machine is too large to be represented by a 32-bit quantity.

Examples

EXAMPLE 1 The usage of the SC_GETNSWP and SC_LIST commands.

The following example demonstrates the usage of the SC_GETNSWP and SC_LIST commands.

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

#define MAXSTRSIZE 80

main(argc, argv)
    int             argc;
    char            *argv[];
{
    swaptbl_t       *s;
    int             i, n, num;
    char            *strtab;    /* string table for path names */

again:
    if ((num = swapctl(SC_GETNSWP, 0)) == -1) {
        perror("swapctl: GETNSWP");
        exit(1);
    }
    if (num == 0) {
        fprintf(stderr, "No Swap Devices Configured\n");
        exit(2);
    }
    /* allocate swaptable for num+1 entries */
    if ((s = (swaptbl_t *)
        malloc(num * sizeof(swapent_t) +
            sizeof(struct swaptable))) ==
        (void *) 0) {
        fprintf(stderr, "Malloc Failed\n");
        exit(3);
    }
    /* allocate num+1 string holders */
    if ((strtab = (char *)
        malloc((num + 1) * MAXSTRSIZE)) == (void *) 0) {
        fprintf(stderr, "Malloc Failed\n");
        exit(3);
    }
    /* initialize string pointers */
    for (i = 0; i < (num + 1); i++) {
        s->swt_ent[i].ste_path = strtab + (i * MAXSTRSIZE);
    }

    s->swt_n = num + 1;
    if ((n = swapctl(SC_LIST, s)) < 0) {
```

EXAMPLE 1 The usage of the SC_GETNSWP and SC_LIST commands. *(Continued)*

```
        perror("swapctl");
        exit(1);
    }
    if (n > num) {        /* more were added */
        free(s);
        free(strtab);
        goto again;
    }
    for (i = 0; i < n; i++)
        printf("%s %ld\n",
            s->swt_ent[i].ste_path, s->swt_ent[i].ste_pages);
}
```

See Also privileges(5)

Name symlink – make a symbolic link to a file

Synopsis

```
#include <unistd.h>

int symlink(const char *name1, const char *name2);
```

Description The `symlink()` function 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()` operation performed on a symbolic link returns the linked-to file, while an `lstat()` operation returns information about the link itself. See `stat(2)`. Unexpected results may occur when a symbolic link is made to a directory. To avoid confusion in applications, the `readlink(2)` call can be used to read the contents of a symbolic link.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned, `errno` is set to indicate the error, and the symbolic link is not made.

Errors The `symlink()` function will fail if:

`EACCES`	Search permission is denied for a component of the path prefix of *name2*.
`EDQUOT`	The directory where the entry for the new symbolic link is being placed cannot be extended because the user's quota of disk blocks on that file system has been exhausted; the new symbolic link cannot be created because the user's quota of disk blocks on that file system has been exhausted; or the user's quota of inodes on the file system where the file is being created has been exhausted.
`EEXIST`	The file referred to by *name2* already exists.
`EFAULT`	The *name1* or *name2* argument points to an illegal address.
`EILSEQ`	The path argument includes non-UTF8 characters and the file system accepts only file names where all characters are part of the UTF-8 character codeset.
`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; the new symbolic link cannot be created because no space is

left on the file system which will contain the link; or 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.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also `cp(1)`, `link(2)`, `open(2)`, `readlink(2)`, `stat(2)`, `unlink(2)`, `attributes(5)`

Name sync – update super block

Synopsis

```
#include <unistd.h>

void sync(void);
```

Description The `sync()` function writes all information in memory that should be on disk, including modified super blocks, modified inodes, and delayed block I/O.

Unlike `fsync(3C)`, which completes the writing before it returns, `sync()` schedules but does not necessarily complete the writing before returning.

Usage The `sync()` function should be used by applications that examine a file system, such as `fsck(1M)`, and `df(1M)`, and is mandatory before rebooting.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also `df(1M)`, `fsck(1M)`, `fsync(3C)`, `attributes(5)`, `standards(5)`

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 The `sysfs()` function returns information about the file system types configured in the system. The number of arguments accepted by `sysfs()` depends on the *opcode* argument, which can take the following values:

`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*, which 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, the value returned depends upon the *opcode* argument as follows:

`GETFSIND` the file-system type index

`GETFSTYP` `0`

`GETNFSTYP` the number of file system types configured

Otherwise, −`1` is returned and `errno` is set to indicate the error.

Errors The `sysfs()` function will fail if:

`EFAULT` The *buf* or *fsname* argument points to an illegal address.

`EINVAL` The *fsname* argument points to an invalid file-system identifier; the *fs_index* argument is 0 or invalid; or the *opcode* argument is invalid.

Name sysinfo – get and set system information strings

Synopsis

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

int sysinfo(int command, char *buf, long count);
```

Description The `sysinfo()` function copies information relating to the operating system on which the process is executing into the buffer pointed to by *buf*. It can also set certain information where appropriate commands are available. The *count* parameter indicates the size of the buffer.

The POSIX P1003.1 interface (see `standards(5)`) `sysconf(3C)` provides a similar class of configuration information, but returns an integer rather than a string.

The values for *command* are as follows:

`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()` 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 might 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 might not be fully qualified. Internet host names can 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 {`PRIV_SYS_ADMIN`} is asserted in the effective set of the calling process.

`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, `sun4u`.

`SI_ARCHITECTURE`
Copy into the array pointed to by *buf* a string describing the basic instruction set architecture of the current system, for example, `sparc`, `mc68030`, `m32100`, or `i386`. These names might not match predefined names in the C language compilation system.

`SI_ARCHITECTURE_64`
Copy into the array pointed to by *buf* a string describing the 64-bit instruction set architecture of the current system, for example, `sparcv9` or `amd64`. These names might not match predefined names in the C language compilation system. This subcode is not recognized on systems that do not allow a 64-bit application to run.

`SI_ARCHITECTURE_32`
Copy into the array pointed to by *buf* a string describing the 32-bit instruction set architecture of the current system, for example, `sparc` or `i386`. These names might not match predefined names in the C language compilation system.

`SI_ARCHITECTURE_K`
Copy into the array pointed to by *buf* a string describing the kernel instruction set architecture of the current system for example `sparcv9` or `i386`. These names might not match predefined names in the C language compilation system.

`SI_ARCHITECTURE_NATIVE`
Copy into the array pointed to by *buf* a string describing the native instruction set architecture of the current system, for example `sparcv9` or `i386`. These names might not match predefined names in the C language compilation system.

`SI_ISALIST`
Copy into the array pointed to by *buf* the names of the variant instruction set architectures executable on the current system.

The names are space-separated and are ordered in the sense of best performance. That is, earlier-named instruction sets might contain more instructions than later-named instruction sets; a program that is compiled for an earlier-named instruction set will most likely run faster on this machine than the same program compiled for a later-named instruction set.

Programs compiled for an instruction set that does not appear in the list will most likely experience performance degradation or not run at all on this machine.

The instruction set names known to the system are listed in `isalist(5)`; these names might not match predefined names or compiler options in the C language compilation system.

This command is obsolete and might be removed in a future release. See `getisax(2)` and the *Linker and Libraries Guide* for a better way to handle instruction set extensions.

`SI_PLATFORM`
Copy into the array pointed to by *buf* a string describing the specific model of the hardware platform, for example, `SUNW,Sun-Blade-1500`, `SUNW,Sun-Fire-T200`, 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. This might be implemented in Read-Only Memory, using software constants set when building the operating system, or by other means, and might contain non-numeric characters. If the function is executed within a non-global zone that emulates a host identifier, then the ASCII representation of the zone's host identifier is copied into the array pointed to by *buf*. 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 not guaranteed to be unique across all vendor's SVR4 implementations and could change over the lifetime of a given system.

`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 {`PRIV_SYS_ADMIN`} is asserted in the effective set of the calling process.

`SI_DHCP_CACHE`
Copy into the array pointed to by *buf* an ASCII string consisting of the ASCII hexidecimal encoding of the name of the interface configured by `boot(1M)` followed by the DHCPACK reply from the server. This command is intended for use only by the `dhcpagent(1M)` DHCP client daemon for the purpose of adopting the DHCP maintenance of the interface configured by `boot`.

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, –1 is returned and `errno` is set to indicate the error.

Errors The `sysinfo()` function will fail if:

`EFAULT` The *buf* argument does not point to a valid address.

`EINVAL` The *count* argument for a non-SET command is less than 0 or the data for a SET command exceeds the limits established by the implementation.

`EPERM` The {`PRIV_SYS_ADMIN`} was not asserted in the effective set of the calling process.

Usage In many cases there is no corresponding programming 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 estimation for *count* is 257, which is likely to cover all strings returned by this interface in typical installations.

See Also `boot(1M)`, `dhcpagent(1M)`, `getisax(2)`, `uname(2)`, `gethostid(3C)`, `gethostname(3C)`, `sysconf(3C)`, `isalist(5)`, `privileges(5)`, `standards(5)`, `zones(5)`

Linker and Libraries Guide

Name time – get time

Synopsis

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

time_t time(time_t *tloc);
```

Description The `time()` function 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. If *tloc* points to an illegal address, `time()` fails and its actions are undefined.

Return Values Upon successful completion, `time()` returns the value of time. Otherwise, `(time_t)−1` is returned and `errno` is set to indicate the error.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `stime(2)`, `ctime(3C)`, `attributes(5)`, `standards(5)`

Name times – get process and child process times

Synopsis

```
#include <sys/times.h>
#include <limits.h>

clock_t times(struct tms *buffer);
```

Description The `times()` function 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;
```

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>`.

The times of a terminated child process are included in the `tms_cutime` and `tms_cstime` members of the parent when `wait`(3C) or `waitpid`(3C) returns the process ID of this terminated child. If a child process has not waited for its children, their times will not be included in its times.

The `tms_utime` member is the CPU time used while executing instructions in the user space of the calling process.

The `tms_stime` member is the CPU time used by the system on behalf of the calling process.

The `tms_cutime` member is the sum of the `tms_utime` and the `tms_cutime` of the child processes.

The `tms_cstime` member 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, since an arbitrary point in the past (for example, system start-up time). This point does not change from one invocation of `times()` within the process to another. The return value may overflow the possible range of type `clock_t`. If `times()` fails, `(clock_t)−1` is returned and `errno` is set to indicate the error.

Errors The `times()` function will fail if:

`EFAULT` The *buffer* argument points to an illegal address.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also time(1), timex(1), exec(2), fork(2), time(2), waitid(2), wait(3C), waitpid(3C), attributes(5), standards(5)

Name uadmin – administrative control

Synopsis

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

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

Description The `uadmin()` function 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. It should be initialized to `NULL` if not used.

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.
- `AD_FASTREBOOT` Bypass BIOS and boot loader

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

`A_DUMP` The system is forced to panic immediately without any further processing and a crash dump is written to the dump device (see `dumpadm(1M)`). 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 subcommands, specified by *fcn*, are available.

`AD_SUSPEND_TO_DISK`
Save the system state to the state file. This subcommand is equivalent to ACPI state S4.

`AD_CHECK_SUSPEND_TO_DISK`
Check if your system supports suspend to disk. Without performing a system suspend/resume, this subcommand checks if this feature is currently available on your system.

`AD_SUSPEND_TO_RAM`
Save the system state to memory This subcommand is equivalent to ACPI state S3.

`AD_CHECK_SUSPEND_TO_RAM`
Check if your system supports suspend to memory. Without performing a system suspend/resume, this subcommand checks if this feature is currently available on your system.

The following subcommands, specified by *fcn*, are obsolete and might be removed in a subsequent release:

`AD_COMPRESS`
Save the system state to the state file with compression of data. This subcommand has been replaced by `AD_SUSPEND_TO_DISK`, which should be used instead.

`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. This subcommand has been replaced by `AD_CHECK_SUSPEND_TO_DISK`, which should be used instead.

`AD_FORCE`
Force `AD_COMPRESS` even when threads of user applications are not suspendable. This subcommand should never be used, as it might result in undefined behavior.

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`.

Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `uadmin()` function will fail if:

`EBUSY`	Suspend is already in progress.
`EINVAL`	The *cmd* argument is invalid.
`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 is not supported on this platform or the command specified by *cmd* is not allowed.

ENXIO Unable to successfully suspend system.

EPERM The {PRIV_SYS_CONFIG} privilege is not asserted in the effective set of the calling process.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	See below.

The A_FREEZE command and its subcommands are Committed.

See Also dumpadm(1M), halt(1M), kernel(1M), reboot(1M), uadmin(1M), attributes(5), privileges(5)

Warnings Shutting down or halting the system by means of uadmin(1M) does not update the boot archive. Avoid using this command after

- editing of files such as /etc/system
- installing new driver binaries or kernel binaries
- updating existing driver binaries or kernel binaries.

Use reboot(1M) or halt(1M) instead.

Name ulimit – get and set process limits

Synopsis

```
#include <ulimit.h>

long ulimit(int cmd, /* newlimit */...);
```

Description The `ulimit()` function provides for control over process limits. It is effective in limiting the growth of regular files. Pipes are limited to `PIPE_MAX` bytes.

The *cmd* values, defined in <`ulimit.h`>, include:

`UL_GETFSIZE` Return the soft 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. The return value is the integer part of the soft file size limit divided by 512. If the result cannot be represented as a `long int`, the result is unspecified.

`UL_SETFSIZE` Set the hard and soft file size limits for output operations of the process to the value of the second argument, taken as a `long int`. Any process may decrease its own hard limit, but only a process with appropriate privileges may increase the limit. The new file size limit is returned. The hard and soft file size limits are set to the specified value multiplied by 512. If the result would overflow an `rlimit_t`, the actual value set is unspecified.

`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.

Return Values Upon successful completion, `ulimit()` returns the value of the requested limit. Otherwise, −1 is returned, the limit is not changed, and `errno` is set to indicate the error.

Errors The `ulimit()` function will fail if:

`EINVAL` The *cmd* argument is not valid.

`EPERM` A process that has not asserted {`PRIV_SYS_RESOURCE`} in its effective set is trying to increase its file size limit.

Usage Since all return values are permissible in a successful situation, an application wishing to check for error situations should set `errno` to 0, then call `ulimit()`, and if it returns −1, check if `errno` is non-zero.

The `getrlimit()` and `setrlimit()` functions provide a more general interface for controlling process limits, and are preferred over `ulimit()`. See `getrlimit(2)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

See Also brk(2), getrlimit(2), write(2), attributes(5), privileges(5), standards(5)

Name umask – set and get file creation mask

Synopsis

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

mode_t umask(mode_t cmask);
```

Description The `umask()` function 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.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `mkdir(1)`, `sh(1)`, `Intro(2)`, `chmod(2)`, `creat(2)`, `mknod(2)`, `open(2)`, `stat.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name umount, umount2 – unmount a file system

Synopsis

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

int umount(const char *file);

int umount2(const char *file, int mflag);
```

Description The `umount()` function requests that a previously mounted file system contained on a block special device or directory be unmounted. The *file* argument is a pointer to the absolute pathname of the file system to be unmounted. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

The `umount2()` function is identical to `umount()`, with the additional capability of unmounting file systems even if there are open files active. The *mflag* argument must contain one of the following values:

0	Perform a normal unmount that is equivalent to `umount()`. The `umount2()` function returns `EBUSY` if there are open files active within the file system to be unmounted.
`MS_FORCE`	Unmount the file system, even if there are open files active. A forced unmount can result in loss of data, so it should be used only when a regular unmount is unsuccessful. The `umount2()` function returns `ENOTSUP` if the specified file systems does not support `MS_FORCE`. Only file systems of type `nfs`, `ufs`, `pcfs`, and `zfs` support `MS_FORCE`.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `umount()` and `umount2()` functions will fail if:

`EACCES`	The permission bits of the mount point do not permit read/write access or search permission is denied on a component of the path prefix. The calling process is not the owner of the mountpoint. The mountpoint is not a regular file or a directory and the caller does not have all privileges available in a its zone. The special device device does not permit read access in the case of read-only mounts or read-write access in the case of read/write mounts.
`EBUSY`	A file on *file* is busy.
`EFAULT`	The file pointed to by *file* points to an illegal address.
`EINVAL`	The file pointed to by *file* is not mounted.
`ELOOP`	Too many symbolic links were encountered in translating the path pointed to by *file*.

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.
ENOENT	The file pointed to by *file* does not exist or is not an absolute path.
ENOLINK	The file pointed to by *file* is on a remote machine and the link to that machine is no longer active.
ENOTBLK	The file pointed to by *file* is not a block special device.
EPERM	The {PRIV_SYS_MOUNT} privilege is not asserted in the effective set of the calling process.
EREMOTE	The file pointed to by *file* is remote.

The umount2() function will fail if:

ENOTSUP	The file pointed to by *file* does not support this operation.

Usage The umount() and umount2() functions can be invoked only by a process that has the {PRIV_SYS_MOUNT} privilege asserted in its effective set.

Because it provides greater functionality, the umount2() function is preferred.

See Also mount(2), privileges(5)

Name uname – get name of current operating system

Synopsis

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

int uname(struct utsname *name);
```

Description The `uname()` function stores information identifying the current operating system in the structure pointed to by *name*.

The `uname()` function uses the `utsname` structure, 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];
```

The `uname()` function returns a null-terminated character string naming the current operating system in the character array `sysname`. Similarly, the `nodename` member contains the name by which the system is known on a communications network. The `release` and `version` members further identify the operating system. The `machine` member contains a standard name that identifies the hardware on which the operating system is running.

Return Values Upon successful completion, a non-negative value is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `uname()` function will fail if:

`EFAULT` The *name* argument points to an illegal address.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `uname(1)`, `sysinfo(2)`, `sysconf(3C)`, `attributes(5)`, `standards(5)`

Name unlink, unlinkat – remove directory entry

Synopsis

```
#include <unistd.h>

int unlink(const char *path);

int unlinkat(int dirfd, const char *path, int flag);
```

Description The `unlink()` function removes a link to a file. If *path* names a symbolic link, `unlink()` removes the symbolic link named by *path* and does not affect any file or directory named by the contents of the symbolic link. Otherwise, `unlink()` removes the link named by the pathname pointed to by *path* and decrements the link count of the file referenced by the link.

The `unlinkat()` function also removes a link to a file. See `fsattr(5)`. If the *flag* argument is 0, the behavior of `unlinkat()` is the same as `unlink()` except in the processing of its *path* argument. If *path* is absolute, `unlinkat()` behaves the same as `unlink()` and the *dirfd* argument is unused. If *path* is relative and *dirfd* has the value `AT_FDCWD`, defined in `<fcntl.h>`, `unlinkat()` also behaves the same as `unlink()`. Otherwise, *path* is resolved relative to the directory referenced by the *dirfd* argument.

If the *flag* argument is set to the value `AT_REMOVEDIR`, defined in `<fcntl.h>`, `unlinkat()` behaves the same as `rmdir(2)` except in the processing of the *path* argument as described above.

When the file's link count becomes 0 and no process has the file open, the space occupied by the file will be freed and the file is no longer accessible. If one or more processes have the file open when the last link is removed, the link is removed before `unlink()` or `unlinkat()` returns, but the removal of the file contents is postponed until all references to the file are closed.

If the *path* argument is a directory and the filesystem supports `unlink()` and `unlinkat()` on directories, the directory is unlinked from its parent with no cleanup being performed. In UFS, the disconnected directory will be found the next time the filesystem is checked with `fsck(1M)`. The `unlink()` and `unlinkat()` functions will not fail simply because a directory is not empty. The user with appropriate privileges can orphan a non-empty directory without generating an error message.

If the *path* argument is a directory and the filesystem does not support `unlink()` and `unlink()` on directories (for example, ZFS), the call will fail with `errno` set to `EPERM`.

Upon successful completion, `unlink()` and `unlinkat()` will mark for update the `st_ctime` and `st_mtime` fields of the parent directory. If the file's link count is not 0, the `st_ctime` field of the file will be marked for update.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned, `errno` is set to indicate the error, and the file is not unlinked.

Errors The `unlink()` and `unlinkat()` functions will fail if:

`EACCES`	Search permission is denied for a component of the *path* prefix, or 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, the user does not own the file, and the user is not a privileged user.
`EBUSY`	The entry to be unlinked is the mount point for a mounted file system.
`EFAULT`	The *path* argument points to an illegal address.
`EILSEQ`	The path argument includes non-UTF8 characters and the file system accepts only file names where all characters are part of the UTF-8 character codeset.
`EINTR`	A signal was caught during the execution of the `unlink()` function.
`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`	The named file does not exist or is a null pathname.
`ENOLINK`	The *path* argument 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 or the provided directory descriptor for `unlinkat()` is not `AT_FDCWD` or does not reference a directory.
`EPERM`	The named file is a directory and {`PRIV_SYS_LINKDIR`} is not asserted in the effective set of the calling process, or the filesystem implementation does not support `unlink()` or `unlinkat()` on directories.
`EROFS`	The directory entry to be unlinked is part of a read-only file system.

The `unlink()` and `unlinkat()` functions may fail if:

`ENAMETOOLONG`	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {`PATH_MAX`}.
`ETXTBSY`	The entry to be unlinked is the last directory entry to a pure procedure (shared text) file that is being executed.

Usage Applications should use `rmdir(2)` to remove a directory.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	`unlink()` is Standard; `unlinkat()` is Evolving
MT-Level	Async-Signal-Safe

See Also `rm(1)`, `close(2)`, `link(2)`, `open(2)`, `rmdir(2)`, `remove(3C)`, `attributes(5)`, `privileges(5)`, `fsattr(5)`

Name ustat – get file system statistics

Synopsis

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

int ustat(dev_t dev, struct ustat *buf);
```

Description The `ustat()` function returns information about a mounted file system. The *dev* argument is a device number identifying a device containing a mounted file system (see `makedev(3C)`). The *buf* argument is a pointer to a `ustat` structure that includes the following members:

```
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 `f_fname` and `f_fpack` members may not contain significant information on all systems; in this case, these members will contain the null character as the first character.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `ustat()` function will fail if:

`ECOMM`	The *dev* argument is on a remote machine and the link to that machine is no longer active.
`EFAULT`	The *buf* argument points to an illegal address.
`EINTR`	A signal was caught during the execution of the `ustat()` function.
`EINVAL`	The *dev* argument is not the device number of a device containing a mounted file system.
`ENOLINK`	The *dev* argument refers to a device on a remote machine and the link to that machine is no longer active.
`EOVERFLOW`	One of the values returned cannot be represented in the structure pointed to by *buf*.

Usage The `statvfs(2)` function should be used in favor of `ustat()`.

See Also `stat(2)`, `statvfs(2)`, `makedev(3C)`, `lfcompile(5)`

Bugs The NFS revision 2 protocol does not permit the number of free files to be provided to the client; therefore, when `ustat()` has completed 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);
```

Description The `utime()` function sets the access and modification times of the file pointed to by *path*, and causes the time of the last file status change (`st_ctime`) to be updated.

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 a process that has the {`PRIV_FILE_OWNER`} privilege asserted in its effective set can use `utime()` in this manner.

The `utimbuf` structure contains the following members:

```
time_t   actime;    /* access time */
time_t   modtime;   /* modification time */
```

The times contained in the members of the `utimbuf` structure are measured in seconds since 00:00:00 UTC, January 1, 1970.

Return Values Upon successful completion, 0 is returned. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors The `utime()` function will fail if:

`EACCES`	Search permission is denied by a component of the *path* prefix.
`EACCES`	The process does not have appropriate privileges and is not the owner of the file, write permission is denied for the file, and *times* is `NULL`.
`EFAULT`	The *path* argument points to an illegal address.
`EINTR`	A signal was caught during the execution of 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*.
`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`	The *path* argument 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 the owner of the file, {PRIV_FILE_OWNER} is not asserted in the effective set of the calling process, and *times* is not NULL.
EROFS	The file system containing the file is mounted read-only.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See standards(5).

See Also futimens(2), stat(2), utimes(2), attributes(5), privileges(5), standards(5)

Name utimes, futimesat – set file access and modification times

Synopsis

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

int utimes(const char *path, const struct timeval times[2]);

int futimesat(int fildes, const char *path,
     const struct timeval times[2]);
```

Description The `utimes()` function sets the access and modification times of the file pointed to by the *path* argument to the value of the *times* argument. It allows time specifications accurate to the microsecond.

The `futimesat()` function also sets access and modification times. See `fsattr(5)`. If *path* is a relative path name, however, `futimesat()` resolves the path relative to the *fildes* argument rather than the current working directory. If *fildes* is set to `AT_FDCWD`, defined in `<fcntl.h>`, `futimesat()` resolves the path relative to the current working directory. If *path* is a null pointer, `futimesat()` sets the access and modification times on the file referenced by *fildes*. The *fildes* argument is ignored even when `futimesat()` is provided with an absolute path.

The *times* argument is an array of `timeval` structures. The first array member represents the date and time of last access, and the second member represents the date and time of last modification. The times in the `timeval` structure are measured in seconds and microseconds since the Epoch, although rounding toward the nearest second may occur.

If the *times* argument is a null pointer, the access and modification times of the file are set to the current time. The effective user ID of the process must be the same as the owner of the file, or must have write access to the file or the {`PRIV_FILE_OWNER`} privilege to use this call in this manner. Upon completion, `utimes()` will mark the time of the last file status change, `st_ctime`, for update.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned, `errno` is set to indicate the error, and the file times will not be affected.

Errors The `utimes()` and `futimesat()` functions will fail if:

`EACCES`	Search permission is denied by a component of the path prefix; or the *times* argument is a null pointer and the effective user ID of the process does not match the owner of the file and write access is denied.
`EFAULT`	The *path* or *times* argument points to an illegal address. For `futimesat()`, *path* might have the value `NULL` if the *fildes* argument refers to a valid open file descriptor.
`EINTR`	A signal was caught during the execution of the `utimes()` function.
`EINVAL`	The number of microseconds specified in one or both of the `timeval` structures pointed to by *times* 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 resolving *path*.
ENAMETOOLONG	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
ENOLINK	The *path* argument points to a remote machine and the link to that machine is no longer active.
ENOENT	A component of *path* does not name an existing file or *path* is an empty string.
ENOTDIR	A component of the path prefix is not a directory or the *path* argument is relative and the *fildes* argument is not AT_FDCWD or does not refer to a valid directory.
EPERM	The *times* argument is not a null pointer and the calling process's effective user ID has write access to the file but does not match the owner of the file and the calling process does not have the appropriate privileges.
EROFS	The file system containing the file is read-only.

The utimes() and futimesat() functions may fail if:

ENAMETOOLONG	Path name resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
Standard	See below.

For utimes(), see standards(5).

See Also

futimens(2), stat(2), utime(2), attributes(5), fsattr(5), standards(5)

Name uucopy – no-fault memory-to-memory copy

Synopsis

```
#include <strings.h>

int uucopy(const void *s1, void *s2, size_t n);
```

Description The `uucopy()` function copies *n* bytes from memory area *s1* to *s2*. Copying between objects that overlap could corrupt one or both buffers.

Unlike `bcopy(3C)`, `uucopy()` does not cause a segmentation fault if either the source or destination buffer includes an illegal address. Instead, it returns −1 and sets `errno` to `EFAULT`. This error could occur after the operation has partially completed, so the contents of the buffer at *s2* are defined if the operation fails.

Return Values Upon successful completion, `uucopy()` returns 0. Otherwise, the function returns −1 and set `errno` to indicate the error.

Errors The `uucopy()` function will fail if:

`EFAULT` Either the *s1* or *s2* arguments points to an illegal address.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	MT-Safe

See Also `bcopy(3C)`, `attributes(5)`

Name vfork, vforkx – spawn new process in a virtual memory efficient way

Synopsis

```
#include <unistd.h>

pid_t vfork(void);

#include <sys/fork.h>

pid_t vforkx(int flags);
```

Description The `vfork()` and `vforkx()` functions create a new process without fully copying the address space of the old process. These functions are useful in instances where the purpose of a `fork(2)` operation is to create a new system context for an `execve()` operation (see `exec(2)`).

Unlike with the `fork()` function, the child process borrows the parent's memory and thread of control until a call to `execve()` or an exit (either abnormally or by a call to `_exit()` (see `exit(2)`). Any modification made during this time to any part of memory in the child process is reflected in the parent process on return from `vfork()` or `vforkx()`. The parent process is suspended while the child is using its resources.

In a multithreaded application, `vfork()` and `vforkx()` borrow only the thread of control that called `vfork()` or `vforkx()` in the parent; that is, the child contains only one thread. The use of `vfork()` or `vforkx()` in multithreaded applications, however, is unsafe due to race conditions that can cause the child process to become deadlocked and consequently block both the child and parent process from execution indefinitely.

The `vfork()` and `vforkx()` functions can normally be used the same way as `fork()` and `forkx()`, respectively. The calling procedure, however, should not return while running in the child's context, since the eventual return from `vfork()` or `vforkx()` in the parent would be to a stack frame that no longer exists. The `_exit()` function should be used in favor of `exit(3C)` if unable to perform an `execve()` operation, since `exit()` will invoke all functions registered by `atexit(3C)` and will flush and close standard I/O channels, thereby corrupting the parent process's standard I/O data structures. Care must be taken in the child process not to modify any global or local data that affects the behavior of the parent process on return from `vfork()` or `vforkx()`, unless such an effect is intentional.

Unlike `fork()` and `forkx()`, fork handlers are not run when `vfork()` and `vforkx()` are called.

The `vfork()` and `vforkx()` functions are deprecated. Their sole legitimate use as a prelude to an immediate call to a function from the exec family can be achieved safely by `posix_spawn(3C)` or `posix_spawnp(3C)`.

Fork Extensions The `vforkx()` function accepts a *flags* argument consisting of a bitwise inclusive-OR of zero or more of the following flags, which are defined in the header `<sys/fork.h>`:

```
FORK_NOSIGCHLD
FORK_WAITPID
```

See fork(2) for descriptions of these flags. If the *flags* argument is 0, vforkx() is identical to vfork().

Return Values Upon successful completion, vfork() and vforkx() return 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, −1 is returned to the parent process, no child process is created, and errno is set to indicate the error.

Errors The vfork() and vforkx() functions will fail if:

EAGAIN The system-imposed limit on the total number of processes under execution (either system-quality or 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.

The vforkx() function will fail if:

EINVAL The *flags* argument is invalid.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Obsolete
MT-Level	Unsafe

See Also exec(2), exit(2), fork(2), ioctl(2), atexit(3C), exit(3C), posix_spawn(3C), posix_spawnp(3C), signal.h(3HEAD), wait(3C), attributes(5), standards(5)

Notes To avoid a possible deadlock situation, processes that are children in the middle of a vfork() or vforkx() are never sent SIGTTOU or SIGTTIN signals; rather, output or ioctls are allowed and input attempts result in an EOF indication.

To forestall parent memory corruption due to race conditions with signal handling, vfork() and vforkx() treat signal handlers in the child process in the same manner as the exec(2) functions: signals set to be caught by the parent process are set to the default action (SIG_DFL) in the child process (see signal.h(3HEAD)). Any attempt to set a signal handler in the child before execve() to anything other than SIG_DFL or SIG_IGN is disallowed and results in setting the handler to SIG_DFL.

On some systems, the implementation of vfork() and vforkx() cause 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() or if <sys/fork.h> is not included in the source calling vforkx().

Name vhangup – virtually "hangup" the current controlling terminal

Synopsis

```
#include <unistd.h>

void vhangup(void);
```

Description The `vhangup()` function is used by the initialization process `init(1M)` (among others) to ensure 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 and revokes 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`). 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 waitid – wait for child process to change state

Synopsis

```
#include <wait.h>

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Description The `waitid()` function suspends the calling process until one of its child processes changes state. It records the current state of a child in the structure pointed to by *infop*. It returns immediately if a child process changed state prior to the call.

The *idtype* and *id* arguments specify which children `waitid()` is to wait for, as follows:

- 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 bitwise OR operation 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(3C)`).

The *infop* argument must point to a `siginfo_t` structure, as defined in `siginfo.h(3HEAD)`. If `waitid()` returns because a child process was found that satisfies the conditions indicated by the arguments *idtype* and *options*, then the structure pointed to by *infop* will be filled by the system with the status of the process. The `si_signo` member will always be equal to `SIGCHLD`.

One instance of a `SIGCHLD` signal is queued for each child process whose status has changed. If `waitid()` returns because the status of a child process is available and `WNOWAIT` was not specified in *options*, any pending `SIGCHLD` signal associated with the process ID of that child process is discarded. Any other pending `SIGCHLD` signals remain pending.

Return Values If `waitid()` returns due to a change of state of one of its children and `WNOHANG` was not used, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error. If `WNOHANG` was used, `0` can be returned (indicating no error); however, no children may have changed state if `info->si_pid` is `0`.

Errors The `waitid()` function will fail if:

`ECHILD` The set of processes specified by *idtype* and *id* does not contain any unwaited processes.

`EFAULT` The *infop* argument points to an illegal address.

`EINTR` The `waitid()` function was interrupted due to the receipt of a signal by the calling process.

`EINVAL` An invalid value was specified for *options*, or *idtype* and *id* specify an invalid set of processes.

Usage With *options* equal to `WEXITED | WTRAPPED`, `waitid()` is equivalent to `waitpid(3C)`. With *idtype* equal to `P_ALL` and *options* equal to `WEXITED | WTRAPPED`, `waitid()` is equivalent to `wait(3C)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `Intro(2)`, `exec(2)`, `exit(2)`, `fork(2)`, `pause(2)`, `sigaction(2)`, `ptrace(3C)`, `signal(3C)`, `siginfo.h(3HEAD)`, `wait(3C)`, `waitpid(3C)`, `attributes(5)`, `standards(5)`

Name write, pwrite, writev – write on a file

Synopsis

```
#include <unistd.h>

ssize_t write(int fildes, const void *buf, size_t nbyte);

ssize_t pwrite(int fildes, const void *buf, size_t nbyte,
     off_t offset);

#include <sys/uio.h>

ssize_t writev(int fildes, const struct iovec *iov, int iovcnt);
```

Description The `write()` function attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor, *fildes*.

If *nbyte* is 0, `write()` will return 0 and have no other results if the file is a regular file; otherwise, the results are unspecified.

On a regular file or other file capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file offset associated with *fildes*. Before successful return from `write()`, the file offset is incremented by the number of bytes actually written. On a regular file, if this incremented file offset is greater than the length of the file, the length of the file will be set to this file offset.

If the `O_SYNC` bit has been set, write I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.

If *fildes* refers to a socket, `write()` is equivalent to `send(3SOCKET)` with no flags set.

On a file not capable of seeking, writing always takes place starting at the current position. The value of a file offset associated with such a device is undefined.

If the `O_APPEND` flag of the file status flags is set, the file offset will be set to the end of the file prior to each write and no intervening file modification operation will occur between changing the file offset and the write operation.

For regular files, no data transfer will occur past the offset maximum established in the open file description with *fildes*.

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).

If `write()` is interrupted by a signal before it writes any data, it will return −1 with `errno` set to `EINTR`.

If `write()` is interrupted by a signal after it successfully writes some data, it will return the number of bytes written.

If `write()` exceeds the process file size limit, the application generates a `SIGXFSZ` signal, whose default behavior is to dump core.

After a `write()` to a regular file has successfully returned:

- Any successful `read(2)` from each byte position in the file that was modified by that write will return the data specified by the `write()` for that position until such byte positions are again modified.
- Any subsequent successful `write()` to the same byte position in the file will overwrite that file data.

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, a FIFO, a socket, or a 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`.

Upon successful completion, where *nbyte* is greater than 0, `write()` will mark for update the `st_ctime` and `st_mtime` fields of the file, and if the file is a regular file, the `S_ISUID` and `S_ISGID` bits of the file mode may be cleared.

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 socket or to a stream that is not a pipe or a 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.

The `write()` and `writev()` functions will fail if the stream head had processed an asynchronous error before the call. In this case, the value of `errno` does not reflect the result of `write()` or `writev()` but reflects the prior error.

`pwrite()` The `pwrite()` function is equivalent to `write()`, except that it writes into a given position and does not change the file offset (regardless of whether `O_APPEND` is set). 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()` The `writev()` function 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. The `writev()` function always writes all data from an area before proceeding to the next.

If *fildes* refers to a regular file and all of the `iov_len` members in the array pointed to by *iov* are 0, `writev()` will return 0 and have no other effect. For other file types, the behavior is unspecified.

If the sum of the `iov_len` values is greater than `SSIZE_MAX`, the operation fails and no data is transferred.

Return Values Upon successful completion, `write()` returns the number of bytes actually written to the file associated with *fildes*. This number is never greater than *nbyte*. Otherwise, −1 is returned, the file-pointer remains unchanged, and `errno` is set to indicate the error.

Upon successful completion, `writev()` returns the number of bytes actually written. Otherwise, it returns −1, the file-pointer remains unchanged, and `errno` is set to indicate an error.

Errors The `write()`, `pwrite()`, and `writev()` functions will fail if:

`EAGAIN` Mandatory file/record locking is set, `O_NDELAY` or `O_NONBLOCK` is set, and there is a blocking record lock; an attempt is made to write to a stream that can not accept data with the `O_NDELAY` or `O_NONBLOCK` flag set; or 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` The *fildes* argument 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.

`EDQUOT` The user's quota of disk blocks on the file system containing the file has been exhausted.

`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)`).

`EFBIG` The file is a regular file, *nbyte* is greater than 0, and the starting position is greater than or equal to the offset maximum established in the file description associated with *fildes*.

EINTR A signal was caught during the write operation and no data was transferred.

EIO The process is in the background and is attempting to write to its controlling terminal whose TOSTOP flag is set, or 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, or the system record lock table was full and the write could not go to sleep until the blocking record lock was removed.

ENOLINK The *fildes* argument 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 streams with insufficient streams memory resources available in the system.

ENXIO A hangup occurred on the stream being written to.

EPIPE An attempt is made to write to a pipe or a FIFO that is not open for reading by any process, or that has only one end open (or to a file descriptor created by socket(3SOCKET), using type SOCK_STREAM that is no longer connected to a peer endpoint). A SIGPIPE signal will also be sent to the thread. The process dies unless special provisions were taken to catch or ignore the signal.

ERANGE The transfer request size was outside the range supported by the streams file associated with *fildes*.

The write() and pwrite() functions will fail if:

EFAULT The *buf* argument points to an illegal address.

EINVAL The *nbyte* argument overflowed an ssize_t.

The pwrite() function fails and the file pointer remains unchanged if:

ESPIPE The *fildes* argument is associated with a pipe or FIFO.

The write() and writev() functions may fail if:

EINVAL The stream or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device.

ENXIO A hangup occurred on the stream being written to.

A write to a streams file may 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.

The `writev()` function may fail if:

`EINVAL` The *iovcnt* argument was less than or equal to 0 or greater than {`IOV_MAX`}; one of the `iov_len` values in the *iov* array was negative; or the sum of the `iov_len` values in the *iov* array overflowed an `ssize_t`.

Usage The `pwrite()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	`write()` is Async-Signal-Safe
Standard	See `standards(5)`.

See Also `Intro(2)`, `chmod(2)`, `creat(2)`, `dup(2)`, `fcntl(2)`, `getrlimit(2)`, `ioctl(2)`, `lseek(2)`, `open(2)`, `pipe(2)`, `ulimit(2)`, `send(3SOCKET)`, `socket(3SOCKET)`, `attributes(5)`, `lf64(5)`, `standards(5)`, `streamio(7I)`

Name yield – yield execution to another lightweight process

Synopsis

```
#include <unistd.h>

void yield(void);
```

Description The `yield()` function 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`(3C)