## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>9</td>
</tr>
<tr>
<td>Introduction</td>
<td>15</td>
</tr>
<tr>
<td>Intro(2)</td>
<td>16</td>
</tr>
<tr>
<td>System Calls</td>
<td>39</td>
</tr>
<tr>
<td>access(2)</td>
<td>40</td>
</tr>
<tr>
<td>acct(2)</td>
<td>42</td>
</tr>
<tr>
<td>acl(2)</td>
<td>43</td>
</tr>
<tr>
<td>adjtime(2)</td>
<td>45</td>
</tr>
<tr>
<td>alarm(2)</td>
<td>46</td>
</tr>
<tr>
<td>audit(2)</td>
<td>48</td>
</tr>
<tr>
<td>auditon(2)</td>
<td>49</td>
</tr>
<tr>
<td>auditsvc(2)</td>
<td>53</td>
</tr>
<tr>
<td>brk(2)</td>
<td>55</td>
</tr>
<tr>
<td>chdir(2)</td>
<td>57</td>
</tr>
<tr>
<td>chmod(2)</td>
<td>59</td>
</tr>
<tr>
<td>chown(2)</td>
<td>62</td>
</tr>
<tr>
<td>chroot(2)</td>
<td>65</td>
</tr>
<tr>
<td>close(2)</td>
<td>67</td>
</tr>
<tr>
<td>creat(2)</td>
<td>69</td>
</tr>
<tr>
<td>dup(2)</td>
<td>71</td>
</tr>
<tr>
<td>exec(2)</td>
<td>72</td>
</tr>
<tr>
<td>exit(2)</td>
<td>78</td>
</tr>
<tr>
<td>fcntl(2)</td>
<td>81</td>
</tr>
</tbody>
</table>
mprotect(2) 160
msgctl(2) 161
msgget(2) 163
msgids(2) 164
msgrecv(2) 166
msgsnp(2) 168
msgsnd(2) 171
munmap(2) 173
nice(2) 174
ntp_adjtime(2) 175
ntp_gettime(2) 176
open(2) 177
pause(2) 183
pcsample(2) 184
pipe(2) 185
poll(2) 186
p_online(2) 189
priocntl(2) 191
priocntlset(2) 200
processor_bind(2) 202
processor_info(2) 204
profil(2) 205
pset_bind(2) 207
pset_create(2) 209
pset_info(2) 211
ptrace(2) 212
putmsg(2) 214
read(2) 217
readlink(2) 222
rename(2) 223
resolvepath(2) 226
rmdir(2) 227
semctl(2) 229
semget(2) 232
semids(2) 234
semop(2) 236
setpgid(2) 239
setpgid(2) 240

Contents 5
vhangup(2)  313
wait(2)  314
waitid(2)  316
waitpid(2)  318
write(2)  320
yield(2)  326

Index  327
Preface

Both novice users and those familiar 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 6 contains available games and demos.
- Section 7 describes various special files that refer to specific hardware peripherals and device drivers. STREAMS software drivers, modules and the STREAMS-generic set of system calls are also described.
Section 9 provides reference information needed to write device drivers in the kernel environment. It describes two device driver interface specifications: the Device Driver Interface (DDI) and the Driver/Kernel Interface (DKI).

Section 9E describes the DDI/DKI, DDI-only, and DKI-only entry-point routines a developer 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 section 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,
exemple#. 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.
Introduction
**NAME**
Intro – introduction to system calls and error numbers

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

**DESCRIPTION**
This section describes all of the system calls. Most of these calls 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 `_REENTRANT` flag must be defined on the command line at compilation time (`-D_REENTRANT`). When the `_REENTRANT` flag is defined, `errno` becomes a macro which enables each thread to have its own `errno`. This `errno` macro can be used on either side of the assignment, just as if it were a variable.

Applications should use bound threads rather than the `_lwp_*()` functions (see `thr_create(3THR)`). Using LWPs (lightweight processes) directly is not advised because libraries are only safe to use with threads, not LWPs.

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>`.

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 EPERM</td>
<td>Not superuser</td>
</tr>
<tr>
<td></td>
<td>Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or the superuser. It is also returned for attempts by ordinary users to do things allowed only to the superuser.</td>
</tr>
<tr>
<td>2 ENOENT</td>
<td>No such file or directory</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>3 ESRCH</td>
<td>No such process, LWP, or thread</td>
</tr>
<tr>
<td></td>
<td>No process can be found in the system that corresponds to the specified PID, LWPID_t, or thread_t.</td>
</tr>
<tr>
<td>4 EINTR</td>
<td>Interrupted system call</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>
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(2)** function 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. If this error occurs on a resource associated with Remote File Sharing (RFS), it indicates a memory depletion which may be temporary, dependent on system activity at the time the call was invoked.

13 EACCES Permission denied

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

14 EFAULT Bad address

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

15 ENOTBLK Block device required

A non-block device or file was mentioned where a block device was required (for example, in a call to the `mount(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
<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>18 EXDEV</td>
<td>Cross-device link</td>
</tr>
<tr>
<td></td>
<td>A hard link to a file on another device was attempted.</td>
</tr>
<tr>
<td>19 ENODEV</td>
<td>No such device</td>
</tr>
<tr>
<td></td>
<td>An attempt was made to apply an inappropriate operation to a device (for example, read a write-only device).</td>
</tr>
<tr>
<td>20 ENOTDIR</td>
<td>Not a directory</td>
</tr>
<tr>
<td></td>
<td>A non-directory was specified where a directory is required (for example, in a path prefix or as an argument to the <code>chdir(2)</code> function).</td>
</tr>
<tr>
<td>21 EISDIR</td>
<td>Is a directory</td>
</tr>
<tr>
<td></td>
<td>An attempt was made to write on a directory.</td>
</tr>
<tr>
<td>22 EINVAL</td>
<td>Invalid argument</td>
</tr>
<tr>
<td></td>
<td>An invalid argument was specified (for example, unmounting a non-mounted device), mentioning an undefined signal in a call to the <code>signal(3C)</code> or <code>kill(2)</code> function.</td>
</tr>
<tr>
<td>23 ENFILE</td>
<td>File table overflow</td>
</tr>
<tr>
<td></td>
<td>The system file table is full (that is, <code>SYS_OPEN</code> files are open, and temporarily no more files can be opened).</td>
</tr>
<tr>
<td>24 EMFILE</td>
<td>Too many open files</td>
</tr>
<tr>
<td></td>
<td>No process may have more than <code>OPEN_MAX</code> file descriptors open at a time.</td>
</tr>
<tr>
<td>25 ENOTTY</td>
<td>Inappropriate ioctl for device</td>
</tr>
<tr>
<td></td>
<td>A call was made to the <code>ioctl(2)</code> function specifying a file that is not a special character device.</td>
</tr>
<tr>
<td>26 ETXTBSY</td>
<td>Text file busy (obsolete)</td>
</tr>
<tr>
<td></td>
<td>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.)</td>
</tr>
<tr>
<td>27 EFBIG</td>
<td>File too large</td>
</tr>
</tbody>
</table>
### Intro(2)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>28 ENOSPC</td>
<td>No space left on device</td>
</tr>
<tr>
<td>29 ESPIPE</td>
<td>Illegal seek</td>
</tr>
<tr>
<td>30 EROFS</td>
<td>Read-only file system</td>
</tr>
<tr>
<td>31 EMLINK</td>
<td>Too many links</td>
</tr>
<tr>
<td>32 EPIPE</td>
<td>Broken pipe</td>
</tr>
<tr>
<td>33 EDOM</td>
<td>Math argument out of domain of func</td>
</tr>
<tr>
<td>34 ERANGE</td>
<td>Math result not representable</td>
</tr>
<tr>
<td>35 ENOMSG</td>
<td>No message of desired type</td>
</tr>
<tr>
<td>36 EIDRM</td>
<td>Identifier removed</td>
</tr>
</tbody>
</table>

The size of the file exceeded the limit specified by resource RLIMITFSIZE; 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.

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.

A call to the lseek(2) function was issued to a pipe.

An attempt to modify a file or directory was made on a device mounted read-only.

An attempt to make more than the maximum number of links, LINK_MAX, to a file.

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.

The argument of a function in the math package (3M) is out of the domain of the function.

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

An attempt was made to receive a message of a type that does not exist on the specified message queue (see msgrecv(2)).
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)`).

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>37 ECHRNG</td>
<td>Channel number out of range</td>
</tr>
<tr>
<td>38 EL2NSYNC</td>
<td>Level 2 not synchronized</td>
</tr>
<tr>
<td>39 EL3HLT</td>
<td>Level 3 halted</td>
</tr>
<tr>
<td>40 EL3RST</td>
<td>Level 3 reset</td>
</tr>
<tr>
<td>41 ELNRNG</td>
<td>Link number out of range</td>
</tr>
<tr>
<td>42 EUNATCH</td>
<td>Protocol driver not attached</td>
</tr>
<tr>
<td>43 ENOCSI</td>
<td>No CSI structure available</td>
</tr>
<tr>
<td>44 EL2HLT</td>
<td>Level 2 halted</td>
</tr>
<tr>
<td>45 EDEADLK</td>
<td>Deadlock condition</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>46 ENOLCK</td>
<td>No record locks available</td>
</tr>
</tbody>
</table>

There are no more locks available. The system lock table is full (see `fcntl(2)`).

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>47 ECANCELED</td>
<td>Operation canceled</td>
</tr>
</tbody>
</table>

The associated asynchronous operation was canceled before completion.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 ENOTSUP</td>
<td>Not supported</td>
</tr>
</tbody>
</table>

This version of the system does not support this feature. Future versions of the system may provide support.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>49 EDQUOT</td>
<td>Disc quota exceeded</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>58-59</td>
<td>Reserved</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>60</td>
<td>ENOSTR Device not a stream</td>
</tr>
<tr>
<td></td>
<td>A putmsg(2) or getmsg(2) call was attempted on a file descriptor that is not a STREAMS device.</td>
</tr>
<tr>
<td>61</td>
<td>ENODATA No data available</td>
</tr>
<tr>
<td>62</td>
<td>ETIME Timer expired</td>
</tr>
<tr>
<td></td>
<td>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(3THR).</td>
</tr>
<tr>
<td>63</td>
<td>ENOSR Out of stream resources</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>64</td>
<td>ENONET Machine is not on the network</td>
</tr>
<tr>
<td></td>
<td>This error is Remote File Sharing (RFS) specific. It occurs when users try to advertise, unadvertise, mount, or unmount remote resources while the machine has not done the proper startup to connect to the network.</td>
</tr>
<tr>
<td>65</td>
<td>ENOPKG Package not installed</td>
</tr>
<tr>
<td></td>
<td>This error occurs when users attempt to use a call from a package which has not been installed.</td>
</tr>
<tr>
<td>66</td>
<td>EREMOTE Object is remote</td>
</tr>
<tr>
<td></td>
<td>This error is RFS-specific. It occurs when users try to advertise a resource which is not on the local machine, or try to mount/unmount a device (or pathname) that is on a remote machine.</td>
</tr>
<tr>
<td>67</td>
<td>ENOLINK Link has been severed</td>
</tr>
<tr>
<td></td>
<td>This error is RFS-specific. It occurs when the link (virtual circuit) connecting to a remote machine is gone.</td>
</tr>
<tr>
<td>68</td>
<td>EADV Advertise error</td>
</tr>
</tbody>
</table>

Intro(2)
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>69 ESRMNT</td>
<td>Srnmount error</td>
</tr>
<tr>
<td>70 ECOMM</td>
<td>Communication error on send</td>
</tr>
<tr>
<td>71 EPROTO</td>
<td>Protocol error</td>
</tr>
<tr>
<td>76 EDOTDOT</td>
<td>Error 76</td>
</tr>
<tr>
<td>77 EBADMSG</td>
<td>Not a data message</td>
</tr>
<tr>
<td>78 ENAMETOOLONG</td>
<td>File name too long</td>
</tr>
<tr>
<td>79 EOVERFLOW</td>
<td>Value too large for defined data type</td>
</tr>
</tbody>
</table>

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

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

This error is RFS-specific. It occurs when the current process is waiting for a message from a remote machine, and the virtual circuit fails.

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

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

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.

The length of the path argument exceeds PATH_MAX, or the length of a path component exceeds NAME_MAX while _POSIX_NO_TRUNC is in effect; see limits(4).
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, Volume 3
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
<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>91 ESTART</td>
<td>Restartable system call</td>
</tr>
<tr>
<td></td>
<td>Interrupted system call should be restarted.</td>
</tr>
<tr>
<td>92 ESTRPIPE</td>
<td>If pipe/FIFO, don’t sleep in stream head</td>
</tr>
<tr>
<td></td>
<td>Streams pipe error (not externally visible).</td>
</tr>
<tr>
<td>93 ENOTEMPTY</td>
<td>Directory not empty</td>
</tr>
<tr>
<td>94 EUSERS</td>
<td>Too many users</td>
</tr>
<tr>
<td>95 ENOTSOCK</td>
<td>Socket operation on non-socket</td>
</tr>
<tr>
<td>96 EDESTADDRREQ</td>
<td>Destination address required</td>
</tr>
<tr>
<td></td>
<td>A required address was omitted from an operation on a transport endpoint. Destination address required.</td>
</tr>
<tr>
<td>97 EMGSIZE</td>
<td>Message too long</td>
</tr>
<tr>
<td></td>
<td>A message sent on a transport provider was larger than the internal message buffer or some other network limit.</td>
</tr>
<tr>
<td>98 EPROTOTYPE</td>
<td>Protocol wrong type for socket</td>
</tr>
<tr>
<td></td>
<td>A protocol was specified that does not support the semantics of the socket type requested.</td>
</tr>
<tr>
<td>99 ENOPROTOOPT</td>
<td>Protocol not available</td>
</tr>
<tr>
<td></td>
<td>A bad option or level was specified when getting or setting options for a protocol.</td>
</tr>
<tr>
<td>120 EPROTONOSUPPORT</td>
<td>Protocol not supported</td>
</tr>
<tr>
<td></td>
<td>The protocol has not been configured into the system or no implementation for it exists.</td>
</tr>
<tr>
<td>121 ESOCKTNOSUPPORT</td>
<td>Socket type not supported</td>
</tr>
<tr>
<td></td>
<td>The support for the socket type has not been configured into the system or no implementation for it exists.</td>
</tr>
<tr>
<td>122 EOPNOTSUPP</td>
<td>Operation not supported on transport endpoint</td>
</tr>
<tr>
<td></td>
<td>For example, trying to accept a connection on a datagram transport endpoint.</td>
</tr>
<tr>
<td>123 EPFNOSUPPORT</td>
<td>Protocol family not supported</td>
</tr>
</tbody>
</table>
The protocol family has not been configured into the system or no implementation for it exists. Used for the Internet protocols.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>124 EAFNOSUPPORT</td>
<td>Address family not supported by protocol family</td>
</tr>
<tr>
<td></td>
<td>An address incompatible with the requested protocol was used.</td>
</tr>
<tr>
<td>125 EADDRINUSE</td>
<td>Address already in use</td>
</tr>
<tr>
<td></td>
<td>User attempted to use an address already in use, and the protocol does not allow this.</td>
</tr>
<tr>
<td>126 EADDRNOTAVAIL</td>
<td>Cannot assign requested address</td>
</tr>
<tr>
<td></td>
<td>Results from an attempt to create a transport endpoint with an address not on the current machine.</td>
</tr>
<tr>
<td>127 ENETDOWN</td>
<td>Network is down</td>
</tr>
<tr>
<td></td>
<td>Operation encountered a dead network.</td>
</tr>
<tr>
<td>128 ENETUNREACH</td>
<td>Network is unreachable</td>
</tr>
<tr>
<td></td>
<td>Operation was attempted to an unreachable network.</td>
</tr>
<tr>
<td>129 ENETRESET</td>
<td>Network dropped connection because of reset</td>
</tr>
<tr>
<td></td>
<td>The host you were connected to crashed and rebooted.</td>
</tr>
<tr>
<td>130 ECONNABORTED</td>
<td>Software caused connection abort</td>
</tr>
<tr>
<td></td>
<td>A connection abort was caused internal to your host machine.</td>
</tr>
<tr>
<td>131 ECONNRESET</td>
<td>Connection reset by peer</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>132 ENOBUFS</td>
<td>No buffer space available</td>
</tr>
<tr>
<td></td>
<td>An operation on a transport endpoint or pipe was not performed because the system lacked sufficient buffer space or because a queue was full.</td>
</tr>
<tr>
<td>133 EISCONN</td>
<td>Transport endpoint is already connected</td>
</tr>
<tr>
<td>Error Code</td>
<td>Error Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>134 ENOTCONN</td>
<td>Transport endpoint is not connected</td>
</tr>
<tr>
<td>134 ENOTCONN</td>
<td>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.</td>
</tr>
<tr>
<td>143 ESHUTDOWN</td>
<td>Cannot send after transport endpoint shutdown</td>
</tr>
<tr>
<td>143 ESHUTDOWN</td>
<td>A request to send data was disallowed because the transport endpoint has already been shut down.</td>
</tr>
<tr>
<td>144 ETOOMANYREFS</td>
<td>Too many references: cannot splice</td>
</tr>
<tr>
<td>145 ETIMEDOUT</td>
<td>Connection timed out</td>
</tr>
<tr>
<td>145 ETIMEDOUT</td>
<td>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.</td>
</tr>
<tr>
<td>146 ECONNREFUSED</td>
<td>Connection refused</td>
</tr>
<tr>
<td>146 ECONNREFUSED</td>
<td>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.</td>
</tr>
<tr>
<td>147 EHOSTDOWN</td>
<td>Host is down</td>
</tr>
<tr>
<td>147 EHOSTDOWN</td>
<td>A transport provider operation failed because the destination host was down.</td>
</tr>
<tr>
<td>148 EHOSTUNREACH</td>
<td>No route to host</td>
</tr>
<tr>
<td>148 EHOSTUNREACH</td>
<td>A transport provider operation was attempted to an unreachable host.</td>
</tr>
<tr>
<td>149 EALREADY</td>
<td>Operation already in progress</td>
</tr>
<tr>
<td>149 EALREADY</td>
<td>An operation was attempted on a non-blocking object that already had an operation in progress.</td>
</tr>
<tr>
<td>150 EINPROGRESS</td>
<td>Operation now in progress</td>
</tr>
</tbody>
</table>
An operation that takes a long time to complete (such as a \texttt{connect(1)} 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 \texttt{termios(7)}.

**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 \texttt{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 \texttt{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 is superuser.
- 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 (0070) of the file mode is set.

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(4)), it shall be considered an error condition in that implementation. Otherwise, the implementation shall use the first NAME_MAX bytes of the pathname component.

**Foreground Process Group**

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

**{IOV_MAX}**

Maximum number of entries in a struct iovec array.

**{LIMIT}**

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

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.

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:

```c
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:

```c
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.

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 effective user ID of the process is superuser.
- 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.
**Intro(2)**

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Orphaned Process Group</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Path Name</strong></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>A slash by itself names the root directory.</td>
</tr>
<tr>
<td></td>
<td>Unless specifically stated otherwise, the null path name is treated as if it named a non-existent file.</td>
</tr>
<tr>
<td><strong>Process ID</strong></td>
<td>Each process in the system is uniquely identified during its lifetime by a positive integer called a process ID. A process ID may not be reused by the system until the process lifetime, process group lifetime, and session lifetime ends for any process ID, process group ID, and session ID equal to that process ID. Within a process, there are threads with thread id’s, called thread_t and LWID_t. These threads are not visible to the outside process.</td>
</tr>
<tr>
<td><strong>Parent Process ID</strong></td>
<td>A new process is created by a currently active process (see <code>fork(2)</code>). The parent process ID of a process is the process ID of its creator.</td>
</tr>
<tr>
<td><strong>Privilege</strong></td>
<td>Having appropriate privilege means having the capability to override system restrictions.</td>
</tr>
<tr>
<td><strong>Process Group</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Process Group Leader</strong></td>
<td>A process group leader is a process whose process ID is the same as its process group ID.</td>
</tr>
<tr>
<td><strong>Process Group ID</strong></td>
<td>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 <code>kill(2)</code>).</td>
</tr>
<tr>
<td><strong>Process Lifetime</strong></td>
<td>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 <code>wait(2)</code>.</td>
</tr>
<tr>
<td><strong>Process Group Lifetime</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
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)`.

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

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.

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 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)`.

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)`).

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:

```c
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:

```c
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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>semop(2)</strong></td>
<td>READ by user, ALTER by user</td>
</tr>
<tr>
<td><strong>semctl(2)</strong></td>
<td>READ by group, ALTER by group</td>
</tr>
<tr>
<td><strong>semctl(2)</strong></td>
<td>READ by others, ALTER by others</td>
</tr>
</tbody>
</table>

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

- The effective user ID of the process is superuser.
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 (`shm_id`) is a unique positive integer created by a `shmget(2)` call. Each `shm_id` 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 `shm_id_ds` and contains the following members:

```c
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 */
```

The following are descriptions of the `shm_id_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.

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 **shm_id** are granted to a process if one or more of the following are true:

- The effective user ID of the process is superuser.
- The effective user ID of the process matches **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with **shm_id** 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.
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.

**Superuser**

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

**Upstream**

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

**Write Queue**

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

If the requested access is permitted, `access()` succeeds and returns 0. Otherwise, −1 is returned and `errno` is set to indicate the error.

The `access()` function 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`: `path` points to an illegal address.
- `EINTR`: A signal was caught during the `access()` function.
- `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` 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  
*path* points to a remote machine and the link to that 
machine is no longer active.

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

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

The `access()` function 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.

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**  
intro(2), chmod(2), stat(2), attributes(5)
acct(2)

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 be super-user.

The path argument points to the pathname of the accounting file, whose file format is described on the acct(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 effective user of the calling process is not super-user.

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

SEE ALSO exit(2), signal(3C), acct(3HEAD)
NAME acl, facl – get or set a file’s Access Control List (ACL)

SYNOPSIS

#include <sys/acl.h>

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

DESCRIPTION

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 values for cmd are supported:

| SETACL          | nentries ACL entries, specified in buffer aclbufp, are stored in the file’s ACL. This command can only be executed by a process that has an effective user ID equal to the owner of the file. All directories in the path name must be searchable. |
| GETACL          | Buffer aclbufp is filled with the file’s ACL entries. Read access to the file is not required, but all directories in the path name must be searchable. |
| GETACLNT        | The number of entries in the file’s ACL is returned. Read access to the file is not required, but all directories in the path name must be searchable. |

RETURN VALUES

Upon successful completion, acl() and facl() return 0 if cmd is SETACL. If cmd is GETACL or GETACLNT, 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:

| EACCESS          | 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, or GETACLNT; the cmd argument is SETACL and nentries is less than 3; or the cmd argument is SETACL and the ACL specified in aclbufp is not valid. |
| EIO              | A disk I/O error has occurred while storing or retrieving the ACL. |
| 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. |
| ENOTDIR          | A component of the path specified by pathp is not a directory, or the cmd argument is SETACL and an attempt is made to set a default ACL on a file type other than a directory. |

System Calls
ENOSYS The cmd argument is 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.

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

ERofs The cmd argument is 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:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

SEE ALSO getfacl(1), setfacl(1), aclcheck(3SEC), aclsort(3SEC)
adjtime(2)

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 the super-user may adjust the time of day.

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

RETURN VALUES
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 effective user of the calling process is not super-user.

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

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

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

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

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

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

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

The semantic for Solaris threads will move to the per-process semantic specified by POSIX (see `standards(5)`) at this future date. New applications should not rely on the per-thread semantic of `alarm()`, since this semantic will become obsolete.

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

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

The `alarm()` function returns the amount of time previously remaining in the alarm clock of the calling process.
ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

exec(2), fork(2), pause(2), signal(3C), thr_sigsetmask(3THR), attributes(5), standards(5)
The *audit*() function is used to write a record to the system audit log. The data pointed to by *record* is written to the log after a minimal consistency check, with the *length* parameter specifying the size of the record in bytes. The data should be a well-formed audit record as described by *audit.log*(4).

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

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

The *audit*() function will fail if:

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

Only the super-user may successfully execute this call.

The functionality described in this man page is available only if the Basic Security Module (BSM) has been enabled. See *bsmconv*(1M) for more information.
auditon(2)

NAME
   auditon – manipulate auditing

SYNOPSIS
   cc [ flag ... ] file ... -bsm -lsocket -lnsl -lintl [ library ... ]
   #include <sys/param.h>
   #include <bsm/audit.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
                   long pointed to by data. The following values may be returned:
                   
                   AUC_AUDITING   Auditing has been turned on.
                   AUC_NOAUDIT    Auditing has been turned off.
                   AUC_DISABLED   Auditing package installed, not turned on.

   A_SETCOND       Set the system’s audit on/off condition to the value in the integer
                   long pointed to by data. The BSM audit module must be enabled
                   by bsmconv(1M) before auditing can be turned on. The following
                   audit states may 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 \texttt{A\_GETINFO} may fail if the terminal ID contains a network address longer than 32 bits. In this case, the \texttt{A\_GETINFO\_ADDR} command should be used.

\texttt{A\_GETINFO\_ADDR} Returns the audit ID, preselection mask, terminal ID and audit session ID of the specified process in the \texttt{auditinfo\_addr} structure pointed to by \texttt{data}.

\texttt{A\_SETPMASK} Set the preselection mask of the specified process. The \texttt{data} argument points to the \texttt{auditinfo} structure containing the process ID and the preselection mask. The other fields of the structure are ignored and should be set to \texttt{NULL}.

\texttt{A\_SETUMASK} Set the preselection mask for all processes with the specified audit ID. The \texttt{data} argument points to the \texttt{auditinfo} structure containing the audit ID and the preselection mask. The other fields of the structure are ignored and should be set to \texttt{NULL}.

\texttt{A\_SETSMASK} Set the preselection mask for all processes with the specified audit session ID. The \texttt{data} argument points to the \texttt{auditinfo} structure containing the audit session ID and the preselection mask. The other fields of the structure are ignored and should be set to \texttt{NULL}.

\texttt{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 by \texttt{auditsvc(2)} 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 \texttt{au\_qctrl} structure pointed to by \texttt{data}.

\texttt{A\_SETQCTRL} Set the kernel audit queue control parameters as described above in the \texttt{A\_GETQCTRL} command. The \texttt{data} argument points to the \texttt{au\_qctrl} structure containing the audit queue control parameters. The default and maximum values ‘\texttt{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)

\texttt{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 may be used to anchor an absolute path for a path token generated by an application. The data argument points to a buffer into which the path is copied. The length argument 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_SETFSIZE

Set the maximum size of an audit trail file. When the audit file reaches the designated size, it is closed and a new file started. If the maximum size is unset, the audit trail file generated by auditsvc() will grow to the size of the file system. The data argument points to the au_fstat_t structure containing the maximum audit file size in bytes. The size can not be set less than 0x80000 bytes.

A_GETFSIZE

Return the maximum audit file size and current file size in the au_fstat_t structure pointed to by the data argument.

A_GETPOLICY

Return the audit policy flags in the integer long pointed to by data.

A_SETPOLICY

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

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

AUDIT_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 family of functions (see exec(2)). 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.
Add a sequence token to each audit record. The default action is not to include it.

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

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

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.

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

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, or BSM has not been installed.
- **EPERM**  The process’s effective user ID is not super-user.

The functionality described in this man page is available only if the Basic Security Module (BSM) has been enabled. See `bsmconv(1M)` for more information.
The `auditsvc()` function specifies the audit log file to the kernel. The kernel writes audit records to this file until an exceptional condition occurs and then the call returns. The `fd` argument is a file descriptor that identifies the audit file. Applications should open this file for writing before calling `auditsvc()`.

The `limit` argument specifies the number of free blocks that must be available in the audit file system, and causes `auditsvc()` to return when the free disk space on the audit filesystem drops below this limit. Thus, the invoking program can take action to avoid running out of disk space.

The `auditsvc()` function does not return until one of the following conditions occurs:

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

The `auditsvc()` function will fail if:

- **EAGAIN**
  - The descriptor referred to a stream, was marked for System V-style non-blocking I/O, and no data could be written immediately.

- **EBADF**
  - The `fd` argument is not a valid descriptor open for writing.

- **EBUSY**
  - A second process attempted to perform this call.

- **EFBIG**
  - An attempt was made to write a file that exceeds the process’s file size limit or the maximum file size.

- **EINTR**
  - The call is forced to terminate prematurely due to the arrival of a signal whose SV_INTERRUPT bit in `sv_flags` is set (see `sigvec(3UCB)`). The `signal(3C)` function sets this bit for any signal it catches.

- **EINVAL**
  - Auditing is disabled (see `auditon(2)`), or the `fd` argument does not refer to a file of an appropriate type (regular files are always appropriate.)

- **EIO**
  - An I/O error occurred while reading from or writing to the file system.
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOSPC</td>
<td>The user’s quota of disk blocks on the file system containing the file has been exhausted; audit filesystem space is below the specified limit; or there is no free space remaining on the file system containing the file.</td>
</tr>
<tr>
<td>ENXIO</td>
<td>A hangup occurred on the stream being written to.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The process’s effective user ID is not super-user.</td>
</tr>
<tr>
<td>EWOULDBLOCK</td>
<td>The file was marked for 4.2 BSD-style non-blocking I/O, and no data could be written immediately.</td>
</tr>
</tbody>
</table>

**USAGE**

Only processes with an effective user ID of super-user may execute this call successfully.

**SEE ALSO**

auditd(1M), bsmconv(1M), audit(2), auditon(2), sigvec(3UCB), audit.log (4)

**NOTES**

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

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.

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()`.
chdir(2)

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.
chdir(2)

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

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

ENOTDIR
The open file descriptor filedes does not refer to a directory.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>chdir() is Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
chroot(2), attributes(5)
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 fdex to the bit pattern contained in mode. Access permission bits are interpreted as follows:

- **S_ISUID**: Set user ID on execution.
- **S_ISGID**: 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**: Save text image after execution.
- **S_IRWXU**: Read, write, execute by owner.
- **S_IRUSR**: Read by owner.
- **S_IWUSR**: Write by owner.
- **S_IXUSR**: Execute (search if a directory) by owner.
- **S_IRWXG**: Read, write, execute by group.
- **S_IROTH**: Read by others.
- **S_IWOTH**: Write by others.
- **S_IXOTH**: 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 directory has the set group ID bit set, a given file created within that directory will have the same group ID as the directory, if that group ID is part of the group ID set of the process that created the file. Otherwise, the newly created file’s group ID will be set to the effective group ID of the creating process.

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

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

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

**ERRORS**
The `chmod()` function 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.
- **EINVAL** A signal was caught during execution of the function.
- **EIO** An I/O error occurred while reading from or writing to the file system.
- **ELOOP** Too many symbolic links were encountered in translating `path`.
- **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.
- **EPERM** The effective user ID does not match the owner of the file and is not super-user.
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.
- **EIO** An I/O error occurred while reading from or writing to the file system.
- **EINTR** A signal was caught during execution of the fchmod() function.
- **ENOLINK** The path argument points to a remote machine and the link to that machine is no longer active.
- **EPERM** The effective user ID does not match the owner of the file and the process does not have appropriate privilege.
- **EROFS** The file referred to by fildes resides on a read-only file system.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>chmod() is Async-Signal-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

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

**NOTES**

If you use chmod() to change the file group owner permissions on a file with ACL entries, both the file group owner permissions and the ACL mask are changed to the new permissions. Be aware that the new ACL mask permissions may change the effective permissions for additional users and groups who have ACL entries on the file.
**NAME**

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

**SYNOPSIS**

```c
#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);
```

**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.

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

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

The `chown()` and `lchown()` functions will fail if:
chown(2)

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>Search permission is denied on a component of the path prefix of <code>path</code>.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The <code>path</code> argument points to an illegal address.</td>
</tr>
<tr>
<td>EINTR</td>
<td>A signal was caught during the execution of the <code>chown()</code> or <code>lchown()</code> function.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The <code>group</code> or <code>owner</code> argument is out of range.</td>
</tr>
<tr>
<td>EIO</td>
<td>An I/O error occurred while reading from or writing to the file system.</td>
</tr>
<tr>
<td>ELOOP</td>
<td>Too many symbolic links were encountered in translating <code>path</code>.</td>
</tr>
<tr>
<td>ENAMETOOLOWNG</td>
<td>The length of the <code>path</code> argument exceeds <code>{PATH_MAX}</code>, or the length of a <code>path</code> component exceeds <code>{NAME_MAX}</code> while <code>{_POSIX_NO_TRUNC}</code> is in effect.</td>
</tr>
<tr>
<td>ENOLINK</td>
<td>The <code>path</code> argument points to a remote machine and the link to that machine is no longer active.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>Either a component of the path prefix or the file referred to by <code>path</code> does not exist or is a null pathname.</td>
</tr>
<tr>
<td>ENOTDIR</td>
<td>A component of the path prefix of <code>path</code> is not a directory.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The effective user ID does not match the owner of the file or the process is not the super-user and <code>{_POSIX_CHOWN_RESTRICTED}</code> indicates that such privilege is required.</td>
</tr>
<tr>
<td>EROFS</td>
<td>The named file resides on a read-only file system.</td>
</tr>
</tbody>
</table>

The `fchown()` function will fail if:

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

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>chown() is Async-Signal-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

chgrp(1), chown(1), chmod(2), fpathconf(2), system(4), attributes(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 effective user ID of the process must be super-user 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 should fildes be 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.
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENAME_TOO_LONG</td>
<td>The length of the path argument exceeds PATH_MAX, or the length of a path component exceeds NAME_MAX while <code>_POSIX_NO_TRUNC</code> is in effect.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>The named directory does not exist or is a null pathname.</td>
</tr>
<tr>
<td>ENOLINK</td>
<td>The path argument points to a remote machine and the link to that machine is no longer active.</td>
</tr>
<tr>
<td>ENOTDIR</td>
<td>Any component of the path name is not a directory.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The effective user of the calling process is not super-user.</td>
</tr>
</tbody>
</table>

**SEE ALSO**

chroot(1M), chdir(2)

**WARNINGS**

The only use of `fchroot()` that is appropriate is to change back to the system root.
The `close()` function will deallocate the file descriptor indicated by `fdes`. 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 `fdes` 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(3)`) `fdes` 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 `fdes` 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 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 `fdes` 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 `fdes` refers to the master side of a pseudo-terminal, a `SIGHUP` signal is sent to the process group, 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.
close(2)

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

If "fildes" refers to a socket, close() causes the socket to be destroyed. If the socket is connection-mode, and the SOCK_LINGER option is set for the socket, 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.

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:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO intro(3), creat(2), dup(2), exec(2), fcntl(2), ioctl(2), open(2) pipe(2), fattach(3C), fclose(3C), fdetach(3C), fopen(3C), signal(3C), attributes(5), signal(3HEAD), streamio(7I)
NAME
creat – create a new file or rewrite an existing one
SYNOPSIS
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

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

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

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

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

- If the group ID of the new file does not match the effective group ID or one of the
  supplementary group IDs, the S_ISGID bit is cleared.
- All bits set in the process’s file mode creation mask (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) for
  the values of mode).

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

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

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

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

ERRORS
The creat() function will fail:

EACCES  Search permission is denied on a component of the path prefix; the
  file does not exist and the directory in which the file is to be
  created does not permit writing; or the file exists and write
  permission is denied.

EAGAIN  The file exists, mandatory file/record locking is set, and there are
  outstanding record locks on the file (see chmod(2)).

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
Create a new file in the file system.

**System Errors:**
- **EFAULT** The path argument points to an illegal address.
- **EINTR** A signal was caught during the execution of the `creat()` function.
- **EISDIR** The named file is an existing directory.
- **ELOOP** Too many symbolic links were encountered in translating path.
- **EMFILE** The process has too many open files (see `getrlimit(2)`).
- **ENFILE** The system file table is full.
- **ENOENT** A component of the path prefix does not exist, or the path name is null.
- **ENOLINK** The path argument points to a remote machine and the link to that machine is no longer active.
- **ENOSPC** The file system is out of inodes.
- **ENOTDIR** A component of the path prefix is not a directory.
- **EOVERFLOW** The file is a large file at the time of `creat()`.
- **EROFS** The named file resides or would reside on a read-only file system.

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

**See Also:**
`chmod(2)`, `close(2)`, `dup(2)`, `fcntl(2)`, `getrlimit(2)`, `lseek(2)`, `open(2)`, `read(2)`, `umask(2)`, `write(2)`, `stat(3HEAD)`, `attributes(5)`, `largefile(5)`, `lfs(5)`
The `dup()` function returns a new file descriptor having the following in common with the original open file descriptor `fdes`:

- 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(fdes)` function call is equivalent to:

```c
fcntl(fdes, F_DUPFD, 0)
```

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.

The `dup()` function will fail if:

- `EBADF` The `fdes` 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 `fdes` argument is on a remote machine and the link to that machine is no longer active.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO `close(2), creat(2), exec(2), fcntl(2), getrlimit(2), open(2), pipe(2), dup2(3C), lockf(3C), attributes(5)`
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[], char *envp[]);
```

where `argc` is the argument count, `argv` is an array of character pointers to the arguments themselves, and `envp` is an array of character pointers to the environment strings. The `argv` and `envir` arrays are each terminated by a null pointer. The null pointer terminating the `argv` array is not counted in `argc`. As indicated, `argc` is at least one and the first member of the array points to a string containing the name of the file.

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, execvp() and execv() use the contents of that file as standard input to the shell. In this case, the shell becomes the new process image. In a standard-conforming application (see standards(5)), the exec family of functions use /usr/xpg4/bin/sh (see ksh(1)); otherwise, they use /usr/bin/sh (see sh(1)).

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 exec(), execv(), execvp(), and execvp(), 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.

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
Signals set to be caught by the calling process image are set to the default action in the new process image (see `signal(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, effective group ID, saved set-user-ID, and saved set-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)`).

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

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(3RT)`.

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

Timers created by the calling process with `timer_create(3RT)` 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
\texttt{mq\_close(3RT)}.

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

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

- nice value (see \texttt{nice(2)})
- scheduler class and priority (see \texttt{priocntl(2)})
- process ID
- parent process ID
- process group ID
- task ID
- supplementary group IDs
- \texttt{semadj} values (see \texttt{semop(2)})
- session membership (see \texttt{exit(2)} and \texttt{signal(3C)})
- real user ID
- real group ID
- project ID
- trace flag (see \texttt{ptrace(2)} request 0)
- time left until an alarm clock signal (see \texttt{alarm(2)})
- current working directory
- root directory
- file mode creation mask (see \texttt{umask(2)})
- file size limit (see \texttt{ulimit(2)})
- resource limits (see \texttt{getrlimit(2)})
- \texttt{tms\_utime}, \texttt{tms\_stime}, \texttt{tms\_cutime}, and \texttt{tms\_cstime} (see \texttt{times(2)})
- file-locks (see \texttt{fcntl(2)} and \texttt{lockf(3C)})
- controlling terminal
- process signal mask (see \texttt{sigprocmask(2)})
- pending signals (see \texttt{sigpending(2)})

A call to any \texttt{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 \texttt{exec} family marks for
update the \texttt{st\_atime} field of the file. If an \texttt{exec} function failed but was able to locate
the \texttt{process\_image\_file}, whether the \texttt{st\_atime} field is marked for update is unspecified.
Should the function succeed, the process image file is considered to have been opened
with \texttt{open(2)}. The corresponding \texttt{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; or the new process file mode denies execute permission.

- **EAGAIN** Total amount of system memory available when reading using raw I/O is temporarily insufficient.

- **EFAULT** An argument points to an illegal address.

- **EINTR** A signal was caught during the 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 execvp() 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 the new process.

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

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>execle() and execve() are Async-Signal-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

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

**WARNINGS**

If a program is setuid to a user ID other than the super-user, and the program is executed when the real user ID is super-user, then the program has some of the powers of a super-user as well.
NAME | exit, _exit – terminate process

SYNOPSIS
#include <stdlib.h>
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. Each function is called as many times as it was registered.

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 output streams, closes all open streams, and removes all files created by tmpfile(3C).

The _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(2), wait3(3C), waitid(2) or waitpid(2), 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(2), wait3(3C), waitid(2) or waitpid(2).
- If the parent process of the calling process is not executing a wait(2), wait3(3C), waitid(2) or waitpid(2), 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(2), wait3(3C), waitid(2) or waitpid(2). 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(3)).
- Each mapped memory object is unmapped.
Each attached shared-memory segment is detached and the value of \texttt{shm_nattach} (see \texttt{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 \texttt{semadj} value (see \texttt{semop}(2)), that value is added to the \texttt{semval} of the specified semaphore.

If the process is a controlling process, the \texttt{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 \texttt{SIGHUP} signal followed by a \texttt{SIGCONT} signal will be sent to each process in the newly-orphaned process group.

If the parent process has set its \texttt{SA_NOCLDWAIT} flag, or set \texttt{SIGCHLD} to \texttt{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 \texttt{UNLOCK} is performed (see \texttt{plock}(3C) and \texttt{memcntl}(2)).

All open named semaphores in the process are closed as if by appropriate calls to \texttt{sem_close(3RT)}. All open message queues in the process are closed as if by appropriate calls to \texttt{mq_close(3RT)}. 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 \texttt{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 \texttt{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 \texttt{acctadm(1M)}), an extended accounting record is written to the extended task accounting file.

These functions do not return.

No errors are defined.

Normally applications should use \texttt{exit()} rather than \texttt{_exit()}.

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>\texttt{_exit()} is Async-Signal Safe</td>
</tr>
</tbody>
</table>
exit(2)

SEE ALSO acctadm(1M), acct(2), close(2), memcntl(2), semop(2), shmget(2),
sigaction(2), times(2), wait(2), waitid(2), waitpid(2), intro(3), atexit(3C),
fclose(3C), mq_close(3RT), plock(3C), tmpfile(3C), wait3(3C), attributes(5),
signal(3HEAD)
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

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

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

The `fcntl()` function may 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 available values for `cmd` are defined in the header `<fcntl.h>`, which 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_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_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_GETFL**
  Get the file status flags and file access modes, defined in `<fcntl.h>`, for the file description associated with `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 are associated with the file description and do not affect other file descriptors that refer to the same file with different open file descriptions.

- **F_SETFL**
  Set the file status flags, defined in `<fcntl.h>`, for the file description associated with `fildes` from the corresponding bits in the third argument, `arg`, taken as type `int`. Bits corresponding to
the file access mode and the oflag values 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_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_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.

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(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.

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.
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:

```c
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 */
long  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:

```c
typedef struct fshare {
    short f_access;
    short f_deny;
    long 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_RDONLY` Set a file share reservation for read-only access.
- `F_WRONLY` Set a file share reservation for write-only access.
- `F_RDWR` 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_RDONLY` Set a file share reservation to deny read access to other processes.
- `F_WRONLY` Set a file share reservation to deny write access to other processes.
- `F_RDWR` Set a file share reservation to deny read and write access to other processes.
- `F_NODeny` 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_GETFD` Value of flags defined in `<fcntl.h>`. The return value will not be negative.
- `F_SETFD` Value other than `-1`.
- `F_GETFL` Value of file status flags and access modes. The return value will not be negative.
- `F_SETFL` Value other than `-1`.
- `F_GETOWN` Value of the socket owner process or process group; this will not be `−1`.
- `F_SETOWN` Value other than `-1`.
- `F_FREESP` Value of `0`.
- `F_GETLK` Value other than `-1`.
- `F_GETLK64` 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_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, or F_SETLKW64 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 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_SETLKW64, F_SETLK64, 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:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal Safe</td>
</tr>
</tbody>
</table>

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), attributes(5), fcntl(3HEAD)

System Interface 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.

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.
fork(2)

NAME
fork, fork1 – create a new process

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

pid_t fork(void);

pid_t fork1(void);

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

- real user ID, real group ID, effective user ID, effective group ID
- environment
- open file descriptors
- close-on-exec flags (see exec(2))
- signal handling settings (that is, SIG_DFL, SIG_IGN, SIG_HOLD, function address)
- supplementary group IDs
- set-user-ID mode bit
- set-group-ID mode bit
- profiling on/off status
- nice value (see nice(2))
- scheduler class (see priocntl(2))
- all attached shared memory segments (see shmat(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

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

- The child process has a unique process ID which does not match any active process group ID.
The child process has a different parent process ID (that is, the process ID of the parent process).

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

Each shared memory segment remains attached and the value of `shm_nattch` is incremented by 1.

All `semadj` values are cleared (see `semop`).

Process locks, text locks, data locks, and other memory locks are not inherited by the child (see `plock(3C) and memcntl`).

The child process’s `tms` structure is cleared: `tms_utime`, `stime`, `cutime`, and `cstime` are set to 0 (see `times`).

The child processes resource utilizations are set to 0; see `getrlimit`. The `it_value` and `it_interval` values for the `ITIMER_REAL` timer are reset to 0; see `getitimer`.

The set of signals pending for the child process is initialized to the empty set.

Timers created by `timer_create` are not inherited by the child process.

No asynchronous input or asynchronous output operations are inherited by the child.

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

---

**Solaris Threads**

In applications that use the Solaris threads API rather than the POSIX threads API (applications linked with `-lthread but not `-lpthread`), `fork()` duplicates in the child process all threads (see `thr_create`) and LWPs in the parent process. The `fork1()` function duplicates only the calling thread (LWP) in the child process.

---

**POSIX Threads**

In applications that use the POSIX threads API rather than the Solaris threads API (applications linked with `-lpthread`, whether or not linked with `-lthread`), a call to `fork()` is like a call to `fork1()`, which replicates only the calling thread. There is no call that forks a child with all threads and LWPs duplicated in the child.

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

---

**fork() Safety**

If a Solaris threads application calls `fork1()` or a POSIX threads application calls `fork()`, and the child does more than simply call `exec()`, there is a possibility of deadlock occurring in the child. The application should use `pthread_atfork` to ensure safety with respect to this deadlock. A Solaris threads application must explicitly link with `-lpthread` to access `pthread_atfork()`. 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()` or `fork1()`. See “MT-Level of Libraries” on the `attributes` manual page.
Upon successful completion, `fork()` and `fork1()` 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()` function will fail if:

- **EAGAIN** The system-imposed limit on the total number of processes under execution by a single user 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.

**Attributes**

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

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td><code>fork()</code> is Async-Signal-Safe</td>
</tr>
</tbody>
</table>

**See Also**

`alarm(2), exec(2), exit(2), fcntl(2), getitimer(2), getrlimit(2), memcntl(2), mmap(2), nice(2), priocntl(2), ptrace(2), semop(2), shmop(2), times(2), umask(2), wait(2), exit(3C), plock(3C), pthread_atfork(3THR), signal(3C), system(3C), thr_create(3THR), timer_create(3RT), attributes(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 (or LWP) in the child that calls `fork1()` must not depend on any resources held by threads (or LWPs) that no longer exist in the child. In particular, locks held by these threads (or LWPs) will not be released.

In a multithreaded process, `fork()` or `fork1()` can cause blocking system calls to be interrupted and return with an `EINTR` error.

The `fork()` and `fork1()` functions suspend all threads in the process before proceeding. Threads that are executing in the kernel and are in an uninterruptible wait cannot be suspended immediately and therefore cause a delay before `fork()` and `fork1()` can complete. During this delay, since all other threads will have already been suspended, the process will appear "hung."
fpathconf(2)

NAME
fpathconf, pathconf – get configurable pathname variables

SYNOPSIS
#include <unistd.h>

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

DESCRIPTION
The fpathconf() and pathconf() functions provide a method for the application to determine the current value of a configurable limit or option 1 (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 fd 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:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value of name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>_POSIX_VDISABLE</td>
<td>_PC_VDISABLE</td>
<td>2</td>
</tr>
<tr>
<td>_POSIX_ASYNC_IO</td>
<td>_PC_ASYNC_IO</td>
<td>8</td>
</tr>
<tr>
<td>_POSIX_PRIO_IO</td>
<td>_PC_PRIO_IO</td>
<td>8</td>
</tr>
<tr>
<td>_POSIX_SYNC_IO</td>
<td>_PC_SYNC_IO</td>
<td>8</td>
</tr>
<tr>
<td>_PC_NO_TRUNC</td>
<td>_POSIX_NO_TRUNC</td>
<td>3,4</td>
</tr>
<tr>
<td>_PC_PIPE_BUF</td>
<td>_POSIX_PIPE_BUF</td>
<td>6</td>
</tr>
<tr>
<td>_PC_NAME_MAX</td>
<td>_POSIX_NAME_MAX</td>
<td>3,4</td>
</tr>
<tr>
<td>_PC_MAX_INPUT</td>
<td>_POSIX_MAX_INPUT</td>
<td>2</td>
</tr>
<tr>
<td>_PC_MAX_CANON</td>
<td>_POSIX_MAX_CANON</td>
<td>2</td>
</tr>
<tr>
<td>_PC_FILESIZEBITS</td>
<td>_POSIX_FILESIZEBITS</td>
<td>3,4</td>
</tr>
<tr>
<td>_PC_LINK_MAX</td>
<td>_POSIX_LINK_MAX</td>
<td>1</td>
</tr>
<tr>
<td>_PC_PATH_MAX</td>
<td>_POSIX_PATH_MAX</td>
<td>4,5</td>
</tr>
</tbody>
</table>

Notes:
1. If path or fd 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.

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 the implementation needs to use path to determine the value of name and the implementation 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 appropriate privileges file specified by path, or path does not exist, pathconf() returns -1 and errno is set to indicate the error.

If the implementation needs to use fildes to determine the value of name and the implementation does not support the association of name with the file specified by fildes, or if fildes is an invalid file descriptor, fpathconf() will return -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 the implementation's <limits.h> or <unistd.h>.

The pathconf() function will fail if:

EINVAL The value of name is not valid.

ELOOP Too many symbolic links were encountered in resolving path.
The `pathconf()` function may fail if:

- **EACCES** Search permission is denied for a component of the path prefix.
- **EINVAL** The implementation does not support an association of the variable `name` with the specified file.
- **ENAMETOOLONG** The length of the `path` argument exceeds `PATH_MAX` or a pathname component is longer than `NAME_MAX`.
- **ENAMETOOLONG** Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `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 will fail if:

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

The `fpathconf()` function may fail if:

- **EBADF** The `fd` argument is not a valid file descriptor.
- **EINVAL** The implementation does not support an association of the variable `name` with the specified file.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td><code>pathconf()</code> is Async-Signal-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`sysconf(3C), limits(4), attributes(5), standards(5)`
**NAME**
gacct, 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 can be examined 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 queuing facility may wish to record which queue a given task or process was submitted to prior to running. The **flags** field determines whether the contents of buf should be treated as raw data or as an embedded **exacct** structure, as described in `<sys/exacct.h>`.

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. 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.
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.

The invoking process lacks sufficient permission to perform the request operation.

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

See also libexacct(3LIB), attributes(5)
The getaudit() function gets the audit ID, the preselection mask, the terminal ID and the audit session ID for the current process.

Note that getaudit() may 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 structure 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 */
```

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

The getaudit() and setaudit() functions will fail if:
getaudit(2)

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

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

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

SEE ALSO  bsmconv(1M), audit(2)

NOTES  The functionality described in this man page is available only if the Basic Security Module (BSM) has been enabled. See bsmconv(1M) for more information.
### NAME
getauid, setauid – get and set user audit identity

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

int getauid(au_id_t *auid);
int setauid(au_id_t *auid);
```

### 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 process’s effective user ID is not super-user.

### USAGE
Only the super-user may successfully execute these calls.

### SEE ALSO
bsmconv(1M), audit(2), getaudit(2)

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

These system calls have been superseded by `getaudit()` and `setaudit()`.
getcontext(2)

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
When a signal handler is executed, the current user context is saved and a new context is created. If the process leaves the signal handler via longjmp(3UCB), then it is unspecified whether the context at the time of the corresponding set jmp(3UCB) call is restored and thus whether future calls to getcontext() will provide an accurate representation of the current context, since the context restored by longjmp(3UCB) may not contain all the information that setcontext() requires. Signal handlers should use siglongjmp(3C) or setcontext() instead.

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.

SEE ALSO
sigaction(2), sigaltstack(2), sigprocmask(2), bsd_signal(3C), makecontext(3C), setjmp(3UCB), sigsetjmp(3C), ucontext(3HEAD)
# getdents(2)

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

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

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

## DESCRIPTION
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(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(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 effective user of the calling process is not super-user.

USAGE  
This function may be invoked only by the super-user.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  
groups(1), chown(2), getuid(2), setuid(2), getgrnam(3C), initgroups(3C),
attributes(5)
## 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**
Decrement in real time. A `SIGALRM` signal is delivered when this timer expires.

In the current and previous releases, when `setitimer(ITIMER_REAL, ...) is
called in a multithreaded process linked with `-lthread` (Solaris threads) or
`-lpthread` (POSIX threads; see `standards(5)`), the resulting `SIGALRM` is sent to
the bound thread that called `setitimer();` `setitimer()` has a per-thread
semantic when called from a bound thread. This semantic will become obsolete in a
future release. The semantic will move to a per-process semantic, with the resulting
`SIGALRM` being sent to the process. The `SIGALRM` so generated is not maskable on
this bound thread by any signal masking function, `pthread_sigmask(3THR),
thr_sigsetmask(3THR), or sigprocmask(2)`. This is a bug that will not be fixed,
since the per-thread semantic will be discontinued in the next release.

Also, calling this routine from an unbound thread is not guaranteed to work as in
the case of bound threads. The resulting `SIGALRM` may be sent to some other
thread (see `alarm(2)`). This is a bug and will not be fixed since the per-thread
semantic is going to be discontinued.
Calling `setitimer(ITIMER_REAL, . . .)` from a process linked with `-lpthread` (POSIX threads) has the same behavior as Solaris threads described above, where a Solaris bound thread is the same as a POSIX thread in system scheduling scope and a Solaris unbound thread is the same as a POSIX thread in local scheduling scope.

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

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

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

**ITIMER_VIRTUAL**
Decrements in process virtual time. It runs only when the process is executing. A `SIGVTALRM` signal is delivered when it expires. (For multithreaded programs see the WARNINGS section below).

**ITIMER_PROF**
Decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by interpreters in statistically profiling the execution of interpreted programs. Each time the `ITIMER_PROF` timer expires, the `SIGPROF` signal is delivered. Because this signal may interrupt in-progress functions, programs using this timer must be prepared to restart interrupted functions. (For multithreaded programs see the WARNINGS section below).

**ITIMER_REALPROF**
Decrements in real time. It is designed to be used for real-time profiling of multithreaded programs. Each time the `ITIMER_REALPROF` timer expires, one counter in a set of counters maintained by the system for each lightweight process (lwp) is incremented. The counter corresponds to the state of the lwp at the time of the timer tick. All lwps executing in user mode when the timer expires are interrupted into system mode. When each lwp resumes execution in user mode, if any of the elements in its set of counters are non-zero, the `SIGPROF` signal is delivered to the lwp. The `SIGPROF` signal is delivered before any other signal except `SIGKILL`. This signal does not interrupt any in-progress function. A `siginfo` structure, defined in `<sys/siginfo.h>`, is associated with the delivery of the `SIGPROF` signal, and includes the following members:

```c
si_tstamp; /* high resolution timestamp */
si_syscall; /* current syscall */
si_nsysarg; /* number of syscall arguments */
si_sysarg[ ]; /* actual syscall arguments */
si_fault; /* last fault type */
si_faddr; /* last fault address */
si_mstate[ ]; /* ticks in each microstate */
```
The enumeration of microstates (indices into si_mstate) is defined in <sys/msacct.h>. (For multithreaded programs see the WARNINGS section below).

**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.

The setitimer() function will fail if:

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

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

alarm(2), sigprocmask(2), gettimeofday(3C), pthread_attr_setscope(3THR), pthread_sigmask(3THR), sleep(3C), sysconf(3C), attributes(5), standards(5)

**WARNINGS**

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

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

```c
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
```

is equivalent to a Solaris bound thread. A POSIX thread with local process scope, created by the call

```c
pthread_attr_setscope(&attr, PTHREAD_SCOPE_PROCESS);
```

is equivalent to a Solaris unbound thread.

**NOTES**

The microseconds field should not be equal to or greater than one second.

The setitimer() function is independent of the alarm() function.
Do not use `setitimer(ITIMER_REAL)` with the `sleep()` routine. A `sleep(3C)` call wipes out knowledge of the user signal handler for `SIGALRM`.

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.
getmsg, getpmsg – get next message off a stream

#include <stropts.h>

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

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

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:

```c
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.
getmsg(2)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The <code>fdes</code> argument is not a valid file descriptor open for reading.</td>
</tr>
<tr>
<td>EBADMSG</td>
<td>Queue message to be read is not valid for <code>getmsg</code>.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>The <code>ctlptr</code>, <code>dataptr</code>, <code>bandp</code>, or <code>flagsp</code> argument points to an illegal address.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>A signal was caught during the execution of the <code>getmsg</code> function.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>A stream is not associated with <code>fdes</code>.</td>
</tr>
<tr>
<td>ENOSTR</td>
<td>An illegal value was specified in <code>flagsp</code>, or the stream referenced by <code>fdes</code> is linked under a multiplexor.</td>
</tr>
</tbody>
</table>

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.

**SEE ALSO**

`intro(2)`, `poll(2)`, `putmsg(2)`, `read(2)`, `write(2)`

*STREAMS Programming Guide*
getpid, getpgrp, getppid, getpgid – get process, process group, and parent process IDs

#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

Upon successful completion, these functions return 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:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

intro(3), exec(2), fork(2), getsid(2), setpgid(2), setpgrp(2), setsid (2), signal(3C), attributes(5)
getrlimit, setrlimit – control maximum system resource consumption

#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 an effective user ID of super-user can raise a hard limit. Both hard and soft limits can be changed in a single call to setrlimit() subject to the constraints described above. Limits may have an “infinite” value of RLIM_INFINITY. The rlp argument is a pointer to struct rlimit that includes the following members:

```c
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:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLIMIT_CORE</td>
<td>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.</td>
</tr>
<tr>
<td>RLIMIT_CPU</td>
<td>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.</td>
</tr>
<tr>
<td>RLIMIT_DATA</td>
<td>The maximum size of a process’s heap in bytes. The brk(2) function will fail with errno set to ENOMEM.</td>
</tr>
<tr>
<td>RLIMITFSIZE</td>
<td>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.</td>
</tr>
<tr>
<td>RLIMIT_NOFILE</td>
<td>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.</td>
</tr>
</tbody>
</table>
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.

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.

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:

<table>
<thead>
<tr>
<th>Limit</th>
<th>Implementation Defined Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLIMIT_FSIZE</td>
<td>FCHR_MAX</td>
</tr>
<tr>
<td>RLIMIT_NOFILE</td>
<td>OPEN_MAX</td>
</tr>
</tbody>
</table>
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 `rhp` 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 the effective user of the calling process is not super-user.

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 `l164(5)`.

**SEE ALSO**

`brk(2)`, `exec(2)`, `fork(2)`, `open(2)`, `sigaltstack(2)`, `ulimit(2)`, `getdtablesize(3C)`, `malloc(3C)`, `signal(3C)`, `sysconf(3C)`, `l164(5)`, `signal(3HEAD)`
getsid(2)

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.

SEE ALSO  
exec(2), fork(2), getpid(2), getpgid(2), setpgid(2), setsid(2)
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.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

See also `intro(2), setuid(2), attributes(5)`
ioct1(2)

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.
   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.

   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.
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOLINK</td>
<td>The <code>fd</code> argument is on a remote machine and the link to that machine is no longer active.</td>
</tr>
<tr>
<td>ENOTTY</td>
<td>The <code>fd</code> argument is not associated with a STREAMS device that accepts control functions.</td>
</tr>
<tr>
<td>ENXIO</td>
<td>The <code>request</code> and <code>arg</code> arguments are valid for this device driver, but the service requested can not be performed on this particular subdevice.</td>
</tr>
<tr>
<td>ENODEV</td>
<td>The <code>fd</code> argument refers to a valid STREAMS device, but the corresponding device driver does not support the <code>ioctl()</code> function.</td>
</tr>
</tbody>
</table>

STREAMS errors are described in `streamio(7I)`.  

**SEE ALSO**  
`streamio(7I), termio(7I)`
### NAME
kill – send a signal to a process or a group of processes

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

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

### 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(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` family, see `exec(2)`) user ID of the receiving process unless the effective user ID of the sending process is super-user, (see `intro(3)`), or `sig` is `SIGCONT` and the sending process has the same session ID as the receiving process.

If `pid` is greater than 0, `sig` will be sent to the process whose process ID is equal to `pid`.

If `pid` is negative but not `(pid_t)−1`, `sig` will be sent to all processes whose process group ID is equal to the absolute value of `pid` and for which the process has permission to send a signal.

If `pid` is 0, `sig` will be sent to all processes excluding special processes (see `intro(3)`) whose process group ID is equal to the process group ID of the sender.

If `pid` is `(pid_t)−1` and the effective user ID of the sender is not super-user, `sig` will be sent to all processes excluding special processes whose real user ID is equal to the effective user ID of the sender.

If `pid` is `(pid_t)−1` and the effective user ID of the sender is super-user, `sig` will be sent to all processes excluding special processes.

### RETURN VALUES
Upon successful completion, 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:

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>The <code>sig</code> argument is not a valid signal number.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The <code>sig</code> argument is <code>SIGKILL</code> and the <code>pid</code> argument is <code>(pid_t)1</code> (that is, the calling process does not have permission to send the signal to any of the processes specified by <code>pid</code>); or the effective user of the calling process does not match the real or saved user and is not super-user, and the calling process is not sending <code>SIGCONT</code> to a process that shares the same session ID.</td>
</tr>
<tr>
<td>ESRCH</td>
<td>No process or process group can be found corresponding to that specified by <code>pid</code>.</td>
</tr>
</tbody>
</table>

### USAGE
The `sigsend(2)` function provides a more versatile way to send signals to processes.
kill(2)

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
kill(1), intro(3), exec(2), getpid(2), gettid(2), setpgid(2), sigaction(2), sigsend(2), signal(3C), attributes(5), signal(3HEAD)
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 pathname 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. The super-user may make multiple links to a directory. Unless the caller is the super-user, the file named by `existing` must not be a directory.

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

Upon successful completion, 0 is returned. Otherwise, -1 is returned, no link is created, and `errno` is set to indicate the error.

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.
- **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.
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.

The *existing or new* argument points to a remote machine and the link to that machine is no longer active.

The directory that would contain the link cannot be extended.

A component of either path prefix is not a directory.

The file named by *existing* is a directory and the effective user of the calling process is not super-user.

The requested link requires writing in a directory on a read-only file system.

The link named by *new* and the file named by *existing* are on different logical devices (file systems).

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**  symlink(2), unlink(2), attributes(5)
llseek

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

offset_t llseek(int fildes, offset_t offset, int whence);

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.

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.

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.

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.

ESPIPE The fildes argument is associated with a pipe or FIFO.

SEE ALSO creat(2), dup(2), fcntl(2), lseek(2), open(2)
The `lseek()` function sets the file pointer associated with the open file descriptor specified by `fd` 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`.

The symbolic constants `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` 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 `fd` 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.

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.

The `lseek()` function will fail if:

- `EBADF` The `fd` argument is not an open file descriptor.
- `EINVAL` The `whence` argument is not `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`; or the `fd` argument is not a remote file descriptor and the resulting file pointer would be negative.
- `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 `fd` argument is associated with a pipe, a FIFO, or a socket.

In multithreaded applications, using `lseek()` in conjunction with a `read()` or `write()` call on a file descriptor shared by more than one thread is not an atomic operation. To ensure atomicity, use `pread()` or `pwrite()`.
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO creat(2), dup(2), fcntl(2), open(2), read(2), write(2), attributes(5), lf64(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:
_lwp_cond_wait(2)

| 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:

| ETIMEDOUT | The time specified in abstime or reltime has passed. |

**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:

```c
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:

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

This 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:

```c
timestruc_t to;
lwp_mutex_t m;
```
EXAMPLE 3 Use the _lwp_cond_reltimedwait() function in a loop testing some condition. (Continued)

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

If `_lwp_create()` is successful and `new_lwp` is not null, the ID of the new LWP is stored in the location pointed to by `new_lwp`.

The `flags` argument specifies additional attributes for the new LWP. The value in `flags` is constructed by the bitwise inclusive OR operation of the following values:

- **LWP_DETACHED**: The LWP is created detached.
- **LWP_SUSPENDED**: The LWP is created suspended.
- **__LWP_ASLWP**: The LWP created is the ASLWP (Asynchronous Signals LWP) (see `signal(3HEAD)`). The ASLWP should always be created with all signals blocked. If `__LWP_ASLWP` is specified, then the LWP created is the special, designated LWP that handles signals sent to a multithreaded process (ASLWP). There can be only one ASLWP in a multithreaded process, so the creation of another ASLWP will return EINVAL. It should never exit by way of `_lwp_exit(2)` or `exit(2)`. This is a reserved flag and should not be used by any user program. It is documented here for the sake of completion and not for use by an application.

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

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

Upon successful completion, 0 is returned. A non-zero value indicates an error.

If any of the following conditions are detected, `_lwp_create()` fails and returns the corresponding value:
_lwp_create(2)

EFAULT Either the context parameter or the new_lwp parameter point to invalid addresses.

EAGAIN A system limit is exceeded, (for example, too many LWP were created for this real user ID).

EINVAL The __LWP_ASLWP flag was used to create more than one ASLWP in the process. There can be only one ASLWP within a process.

EXAMPLE 1 How a stack is allocated to a new LWP.

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

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

SEE ALSO _lwp_cond_timedwait(2), _lwp_continue(2), _lwp_exit(2),
_lwp_makecontext(2), _lwp_wait(2), alarm(2), exit(2), poll(2), sleep(3C),
thr_create(3THR), signal(3HEAD), ucontext(3HEAD)

NOTES Applications should use bound threads rather than the _lwp_*() functions (see thr_create(3THR)). Using LWPs directly is not advised because libraries are only safe to use with threads, not LWPs.

In Solaris releases 2.5 through 7, the signal SIGALRM is defined to be per-process. This does not affect the behavior of single-threaded or multithreaded applications. If the application was using LWPs directly, and was relying on alarm(2) or sleep(3C), then the application's behavior might be impacted. The calling LWP will not necessarily be the recipient of the SIGALRM signal when SIGALRM is sent to the process. You might have to use a substitute like poll(2), or _lwp_cond_timedwait(2) to simulate the old per-LWP semantic of SIGALRM.
### NAME
_lwp_exit – terminate the calling LWP

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

void _lwp_exit(void);
```

### DESCRIPTION
The _lwp_exit() function causes the calling LWP to terminate. If it is the last LWP in the process, then the process exits with a status of 0 (see _exit(2)).

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

### SEE ALSO
_lwp_create(2), _lwp_wait(2), exit(2)
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:

```c
typedef struct timestruct {
    int tv_nsec;
    int tv_usec;
} timestruct_t;
```

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.

Upon successful completion, \_lwp\_info() returns 0 and fills in the lwpinfo structure pointed to by buffer.

If the following condition is detected, \_lwp\_info() returns the corresponding value:

- EFAULT The buffer argument points to an illegal address.
- EOVERFLOW The size of the tv_sec member of the timestruct_t type pointed to by lwp_utime and lwp_stime is too small to contain the correct number of seconds.

The \_lwp\_info(2) man pages section 2: System Calls • Last Revised 25 Sep 1997
### NAME

_lwp_kill – send a signal to a LWP

### SYNOPSIS

```c
#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(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.

### SEE ALSO

`kill(2), sigaction(2), sigprocmask(2), signal(3HEAD)`
_lwp_makecontext(2)

NAME | _lwp_makecontext – initialize an LWP context

SYNOPSIS | #include <sys/types.h>
#include <sys/lwp.h>
#include <ucontext.h>

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

DESCRIPTION | The _lwp_makecontext() function initializes the user context structure pointed to
by ucp. The user context is defined by ucontext(3HEAD). The resulting user context
can be used by _lwp_create(2) for specifying the initial state of the new LWP. The
user context is set up to start executing the function start_routine with a single
argument, arg, and to call _lwp_exit(2) if start_routine returns. The new LWP will use
the storage starting at stack_base and continuing for stack_size bytes as an execution
stack. The initial value in LWP-private memory will be set to private (see
_lwp_setprivate(2)). The signal mask in the user context is not initialized.

SEE ALSO | _lwp_create(2), _lwp_exit(2), _lwp_setprivate(2), ucontext(3HEAD)
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.

Upon successful completion, 0 is returned. A non-zero value indicates an error. 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.

SEE ALSO
_lwp_create(2)
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 incrementated 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.

EINTR
    The function execution was interrupted by a signal or fork(2).

EBUSY
    The function was called on a semaphore with a zero count.

EOVERFLOW
    The value of the sema argument exceeds SEM_VALUE_MAX.

SEE ALSO

fork(2)
NAME

_lwp_setprivate, _lwp_getprivate – set or get LWP specific storage

SYNOPSIS

#include <sys/lwp.h>

void _lwp_setprivate(void *buffer);
void _lwp_getprivate(void);

DESCRIPTION

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

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

SEE ALSO

_lwp_makecontext(2)
NAME
_lwp_suspend, _lwp_continue – continue or suspend LWP execution

SYNOPSIS
#include <sys/lwp.h>

int _lwp_suspend(lwpid_t target_lwp);
int _lwp_continue(lwpid_t target_lwp);

DESCRIPTION
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.

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

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

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

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

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

Upon successful completion, 0 is returned. A non-zero value indicates an error.

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

EINTR The _lwp_wait() function was interrupted by a signal.

EDEADLK All LWPs in this process would be blocked waiting for LWPs to
terminate, or the calling LWP is attempting to wait for itself.

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

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

SEE ALSO _lwp_create(2), _lwp_exit(2)
memcntl – memory management control

**SYNOPSIS**

```c
#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.

At present arg is unused, but must be 0 to ensure compatibility with potential future enhancements.

**MC_LOCKAS**

Lock in memory all pages mapped by the address space with attributes attr. At present addr and len are unused, but must be NULL and 0 respectively, to ensure compatibility with potential future enhancements. 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 returns immediately once all write operations are scheduled; with MS_SYNC the function will not return until all write operations are completed.

MS_INVALIDATE invalidates all cached copies of data in memory, so that further references to the pages will be obtained by the system from their backing storage locations. This operation should be used by applications that require a memory object to be in a known state.
Unlock all pages in the range with attributes attr. At present arg is unused, but must be 0 to ensure compatibility with potential future enhancements.

Remove address space memory locks, and locks on all pages in the address space with attributes attr. At present addr, len, and arg are unused, but must be NULL, 0, and 0 respectively, to ensure compatibility with potential future enhancements.

The mask argument must be 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, all operations, with the exception of MC_SYNC, are restricted to processes with super-user effective user ID.

The memcntl() function subsumes the operations of plock(3C) and mctl(3UCB).

Upon successful completion, memcntl() returns 0; otherwise, it returns -1 and sets errno to indicate an error.

The memcntl() function will fail if:

EAGAIN Some or all of the memory identified by the operation could not be locked when MC_LOCK or MC_LOCKAS was specified.

EBUSY 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; or the arg argument is not valid for the function specified.

ENOMEM 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 process’s effective user ID is not super-user and MC_LOCK, MC_LOCKAS, MC_UNLOCK, or MC_UNLOCKAS was specified.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>
SEE ALSO

fork(2) mmap(2), mprotect(2), mctl(3UCB), mlock(3C), mlockall(3C), msync(3C),
plock(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)
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` is a symbolic link, it is not followed.

The newly created directory is empty with the exception of entries for itself (`.`) and its parent directory (`..`).

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

Upon successful completion, 0 is returned. Otherwise, -1 is returned, no directory is created, and `errno` is set to indicate the error.

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.

**EXIST**
The named file already exists.

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

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

**ELOOP**
Too many symbolic links were encountered in translating `path`. 
The maximum number of links to the parent directory would be exceeded.

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.

A component of the path prefix does not exist or is a null pathname.

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

No free space is available on the device containing the directory.

A component of the path prefix is not a directory.

The path prefix resides on a read-only file system.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

See also chmod(2), mknod(2), umask(2), stat(3HEAD), attributes(5)
mknod(2)

NAME
mknod – make a directory, or a special or ordinary 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 02000 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_IROTH 00040 Read by group.
- S_IWGRP 00020 Write by group.
- S_IXGRP 00010 Execute by group.
- S_IROTH 00007 Read, write, execute (search) by others.
- S_IWOTH 00004 Read 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.
mknod(2)

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

ERofs  The directory in which the file is to be created is located on a read-only file system.

The mknod() function may fail if:

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

USAGE  Normally, applications should use the mkdir(2) routine to make a directory, since the function mknod() may not establish directory entries for the directory itself (.) and the parent directory (..), and appropriate permissions are not required. Similarly, mkfifo(3C) should be used in place of mknod() in order to create FIFOs.

The mknod() function may be invoked only by a privileged user for file types other than FIFO special.

SEE ALSO  chmod(2), creat(2), exec(2), mkdir(2), open(2), stat(2), umask(2), makedev(3C), mkfifo(3C), stat(3HEAD)
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:

\[
\text{pa} = \text{mmap}(\text{addr}, \text{len}, \text{prot}, \text{flags}, \text{fildes}, \text{off});
\]

The \text{mmap()} function establishes a mapping between the address space of the process at an address \text{pa} for \text{len} bytes to the memory object represented by the file descriptor \text{fildes} at offset \text{off} for \text{len} bytes. The value of \text{pa} is a function of the \text{addr} argument and values of \text{flags}, further described below. A successful \text{mmap()} call returns \text{pa} as its result. The address range starting at \text{pa} and continuing for \text{len} bytes will be legitimate for the possible (not necessarily current) address space of the process. The range of bytes starting at \text{off} and continuing for \text{len} bytes will be legitimate for the possible (not necessarily current) offsets in the file or shared memory object represented by \text{fildes}.

The \text{mmap()} function allows \([\text{pa}, \text{pa} + \text{len})\) to extend beyond the end of the object both at the time of the \text{mmap()} and while the mapping persists, such as when the file is created prior to the \text{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 \text{SIGBUS} or \text{SIGSEGV} signal. The \text{mmap()} function cannot be used to implicitly extend the length of files.

The mapping established by \text{mmap()} replaces any previous mappings for those whole pages containing any part of the address space of the process starting at \text{pa} and continuing for \text{len} bytes.

If the size of the mapped file changes after the call to \text{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 \text{mmap()} function is supported for regular files and shared memory objects. Support for any other type of file is unspecified.

The \text{prot} argument determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped. The \text{prot} argument should be either \text{PROT_NONE} or the bitwise inclusive OR of one or more of the other flags in the following table, defined in the header \text{<sys/mman.h>}.  

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROT_READ</td>
<td>Data can be read.</td>
</tr>
<tr>
<td>PROT_WRITE</td>
<td>Data can be written.</td>
</tr>
<tr>
<td>PROT_EXEC</td>
<td>Data can be executed.</td>
</tr>
<tr>
<td>PROT_NONE</td>
<td>Data cannot be accessed.</td>
</tr>
</tbody>
</table>

System Calls 151
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 `fd` is opened with read permission, regardless of the protection options specified. If `PROT_WRITE` is specified, the application must have opened the file descriptor `fd` 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.

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”.

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()`. 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_ANON` is set in `flags`, and `fd` 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 `off` argument is constrained to be aligned and sized according to the value returned by `sysconf(3C)` 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 `fd` 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.
mmap(2)

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.

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 file descriptor is not open for read, regardless of the protection specified, or filedes 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 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 field in flags is invalid (neither MAP_PRIVATE or MAP_SHARED is set).

  The argument len has a value less than or equal to 0.

  MAP_ANON was specified, but the file descriptor was not −1.

- **EMFILE**  
  The number of mapped regions would exceed an implementation-dependent limit (per process or per system).

- **ENODEV**  
  The filedes 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.

Use of MAP_FIXED may result in unspecified behaviour 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 
junction 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 lfs64(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:
mmap(2)

```c
fildes = open(...)
lsseek(fildes, offset, whence)
read(fildes, buf, len)
/* use data in buf */
```

The following is a rewrite using `mmap()`:

```c
fildes = open(...)
address = mmap((caddr_t) 0, len, (PROT_READ | PROT_WRITE),
    MAP_PRIVATE, fildes, offset)
/* use data at address */
```

**SEE ALSO**

close(2), exec(2), fcntl(2), fork(2), getrlimit(2), mprotect(2), munmap(2),
shmat(2), lockf(3C), mlockall(3C), msync(3C), plock(3C), sysconf(3C), 1f64(5)
mount – mount a file system

#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);

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>. 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. The mflag argument is constructed by a bitwise-inclusive-OR of flags from the following list, defined in <sys/mount.h>.

MS_DATA

If this flag is set, 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_OPTIONSTR

If this flag is set, 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_RDONLY**
The file system should be mounted 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_NOSUID**
This option prevents 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_REMOUNT**
Remounts a read-only file system as read-write.

**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_GLOBAL**
Mount a file system globally if the system is configured 
and booted as part of a cluster (see clinfo(1M)).

**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:

**EBUSY**
The dir argument is currently mounted on, is 
someone’s current working directory, or is otherwise 
busy; the device associated with spec is currently 
mounted; or there are no more mount table entries.

**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 or the 
fstype is invalid.

**ELOOP**
Too many symbolic links were encountered in 
translating spec or dir.
**mount(2)**

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENAMETOOLONG</td>
<td>The length of the <code>path</code> argument exceeds <code>PATH_MAX</code>, or the length of a <code>path</code> component exceeds <code>NAME_MAX</code> while <code>_POSIX_NO_TRUNC</code> is in effect.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>None of the named files exists or is a null pathname.</td>
</tr>
<tr>
<td>ENOLINK</td>
<td>The <code>path</code> argument points to a remote machine and the link to that machine is no longer active.</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>The file system state in the super-block is not <code>FSOKAY</code> and <code>mflag</code> requests write permission.</td>
</tr>
<tr>
<td>ENOTBLK</td>
<td>The <code>spec</code> argument is not a block special device.</td>
</tr>
<tr>
<td>ENOTDIR</td>
<td>The <code>dir</code> argument is not a directory, or a component of a path prefix is not a directory.</td>
</tr>
<tr>
<td>ENOTSUP</td>
<td>A global mount is attempted (the <code>MS_GLOBAL</code> flag is set in <code>mflag</code>) on a machine which is not booted as a cluster or a local mount is attempted and <code>dir</code> is within a globally mounted file system.</td>
</tr>
<tr>
<td>ENXIO</td>
<td>The device associated with <code>spec</code> does not exist.</td>
</tr>
<tr>
<td>EOVERFLOW</td>
<td>The length of the option string to be returned in the <code>dataptr</code> argument exceeds the size of the buffer specified by <code>datalen</code>.</td>
</tr>
<tr>
<td>EPERM</td>
<td>The effective user ID is not super-user.</td>
</tr>
<tr>
<td>EREMOTE</td>
<td>The <code>spec</code> argument is remote and cannot be mounted.</td>
</tr>
<tr>
<td>EROFS</td>
<td>The <code>spec</code> argument is write protected and <code>mflag</code> requests write permission.</td>
</tr>
</tbody>
</table>

**USAGE**
The `mount()` function can be invoked only by processes with super-user 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.
mprotect(2)

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.

SEE ALSO mmap(2), plock(3C), mlock(3C), mlockall(3C), sysconf(3C)
msgctl – message control operations

#include <sys/msg.h>

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

The msgctl() function provides a variety of message control operations as specified by cmd. The following cmds 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 cmd can only be executed by a process that has an effective user ID equal to either that of super-user, or to the value of msg_perm.cuid or msg_perm.uid in the data structure associated with msqid. Only super-user can raise the value of msg_qbytes.

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

Upon successful completion, msgctl() returns 0. Otherwise, it returns -1 and sets errno to indicate the error.

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.

- **EPERM**: The cmd argument is IPC_RMID or IPC_SET and the effective user ID of the calling process is not super-user and is not equal to the value of msg_perm.cuid or msg_perm.uid in the data structure associated with msqid.
msgctl(2)

**EPERM**  The *cmd* argument is IPC_SET, an attempt is being made to increase to the value of msg_qbytes, and the effective user ID of the calling process is not super-user.

**EOVERFLOW**  The *cmd* argument is IPC_STAT and *uid* or *gid* is too large to be stored in the structure pointed to by *buf*.

**SEE ALSO**  intro(2), msgget(2), msgrcv(2), msgsnd(2)
**NAME**

msgget – get message queue

**SYNOPSIS**

```c
#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(3)`) are created for `key` if one of the following are true:

- `key` is `IPC_PRIVATE`.
- `key` does not already have a message queue identifier associated with it, and `((msgflg & IPC_CREAT))` is true.

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

- `msg_perm.cuid, msg_perm.uid, msg_perm.cgid, and msg_perm.gid` are set to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of `msg_perm.mode` are set to the low-order 9 bits of `msgflg`.
- `msg_qnum, msg_lspid, msg_lrpid, msg_stime, and msg_rtime` are set to 0.
- `msg_ctime` is set to the current time.
- `msg_qbytes` is set to the system limit.

**RETURN VALUES**

Upon successful completion, a non-negative integer 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(3)`) as specified by the low-order 9 bits of `msgflg` would not be granted.

**EEXIST**

A message queue identifier exists for `key` but `((msgflg & IPC_CREAT))` and `((msgflg & IPC_EXCL))` are both true.

**ENOENT**

A message queue identifier does not exist for `key` and `((msgflg & IPC_CREAT))` is false.

**ENOSPC**

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

**SEE ALSO**

`intro(3), msgctl(2), msgrcv(2), msgsnd(2), ftok(3C)`
msgids(2)

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)):

```c
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 */
        ids = realloc(ids, (nids = n) * sizeof (int));
    }

    for (i = 0; i < n; i++)
```

164 man pages section 2: System Calls • Last Revised 8 Mar 2000
EXAMPLE 1  msgids() example  

(Continued)

    process_msgid(ids[i]);
    free(ids);
}

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  ipcrm(1), ipcs(1), intro(2), msgctl(2), msgget(2), msgsnap(2), msgrcv(2), msgsnd(2), attributes(5)
msgrcv(2)

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 `msgqid` 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*`.

**SEE ALSO** `intro(2)`, `msgctl(2)`, `msgget(2)`, `msgsnd(2)`, `sigaction(2)`
NAME
msgsnap – message queue snapshot operation

SYNOPSIS
#include <sys/msg.h>

msgsnap(int msqid, void *buf, size_t bufsize, 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 bufsize argument specifies the size of buf in bytes. If bufsize is less than
sizeof(msgsnap_head), msgsnap() fails with EINVAL. If bufsize 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)`).

```c
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 */
        mhead = (struct msgsnap_mhead *)
            ((char *)mhead + sizeof(struct msgsnap_mhead) +
             ((mlen + sizeof(size_t) - 1) & -(sizeof(size_t) - 1))));
    }
    free(buf);
}
```
msgsnap(2)

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO | ipcrm(1), ipcs(1), intro(2), msgctl(2), msgget(2), msgids(2), msgsnd(2), attributes(5)
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` 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:

```c
struct mymsg {
    long mtype; /* message type */
    char mtext[1]; /* message text */
}
```

The `mtype` member is a non-zero positive type `long` 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 all queues system-wide is equal to the system-imposed limit.

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.
msgsnd(2)

- `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 `msgflag&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; or 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 *`.

**SEE ALSO**
`intro(2), msgctl(2), msgget(2), msgrcv(2), sigaction(2)`
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.

SEE ALSO

mmap(2), sysconf(3C)
nice(2)

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 \(2 \times \text{NZERO} - 1\) 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.

Only a process with super-user privileges 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.

EPERM  The incr argument is negative or greater than 40 and the effective
user ID of the calling process is not super-user.

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.

SEE ALSO  nice(1), exec(2), priocntl(2)
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.

```c
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) */
};
```

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.

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 user is not super-user.

**SEE ALSO**

`xntpd(1M), ntp_gettime(2)`
The `ntp_gettime()` function reads the local clock value and dispersion, returning the information in `tptr`.

The `ntptimeval` structure contains the following members:

```c
struct ntptimeval {
    struct timeval time; /* current time (ro) */
    int32_t maxerror; /* maximum error (us) (ro) */
    int32_t esterror; /* estimated error (us) (ro) */
};
```

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.

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**

`xntpd(1M), ntp_adjtime(2)"
open(2)

NAME
open – 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 */ ...);

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 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

System Calls 177
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(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 processes executing `open()` naming the same filename in the same directory with `O_EXCL` and `O_CREAT` set. If `O_CREAT` is not set, the effect 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_NONBLOCK or O_NDELAY**  These flags may affect subsequent reads and writes (see `read(2)` and `write(2)`). If both `O_NDELAY` and `O_NONBLOCK` are set, `O_NONBLOCK` 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 process opens the file for writing. An `open()` for writing only blocks until a process 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` will synchronize with later calls to `open() O_WRONLY` 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 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**

If `O_SYNC` is set on a regular file, writes to that file cause the process to block until the data is delivered to the underlying hardware.

**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.

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 *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 *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()` function will fail if:

- **EACCES** Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *oflag* are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created, or `O_TRUNC` is specified and write permission is denied.

- **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.

- **EINTR** A signal was caught during `open()`.

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

- **EIO** The `path` argument names a STREAMS file and a hangup or error occurred during the `open()`.
open(2)

<table>
<thead>
<tr>
<th>ErrorCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EISDIR</td>
<td>The named file is a directory and <em>oflag</em> includes O_WRONLY or O_RDWR.</td>
</tr>
<tr>
<td>ELOOP</td>
<td>Too many symbolic links were encountered in resolving <code>path</code>.</td>
</tr>
<tr>
<td>EMFILE</td>
<td><code>OPEN_MAX</code> file descriptors are currently open in the calling process.</td>
</tr>
<tr>
<td>EMULTIHOP</td>
<td>Components of <code>path</code> require hopping to multiple remote machines and the file system does not allow it.</td>
</tr>
<tr>
<td>ENAMETOOLONG</td>
<td>The length of the <code>path</code> argument exceeds <code>PATH_MAX</code> or a pathname component is longer than <code>NAME_MAX</code>.</td>
</tr>
<tr>
<td>ENFILE</td>
<td>The maximum allowable number of files is currently open in the system.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>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 <code>path</code> argument points to an empty string.</td>
</tr>
<tr>
<td>ENOLINK</td>
<td>The <code>path</code> argument points to a remote machine, and the link to that machine is no longer active.</td>
</tr>
<tr>
<td>ENSR</td>
<td>The <code>path</code> argument names a STREAMS-based file and the system is unable to allocate a STREAM.</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>The directory or file system that would contain the new file cannot be expanded, the file does not exist, and O_CREAT is specified.</td>
</tr>
<tr>
<td>ENOTDIR</td>
<td>A component of the path prefix is not a directory.</td>
</tr>
<tr>
<td>ENXIO</td>
<td>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.</td>
</tr>
<tr>
<td>EOPNOTSUPP</td>
<td>An attempt was made to open a path that corresponds to a <code>AF_UNIX</code> socket.</td>
</tr>
<tr>
<td>EOVERFLOW</td>
<td>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 <code>off_t</code> or O_LARGEFILE is set and the size of the file cannot be represented correctly in an object of type <code>off64_t</code>.</td>
</tr>
<tr>
<td>EROFS</td>
<td>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 <code>oflag</code> argument.</td>
</tr>
</tbody>
</table>

The `open()` function may fail if:

<table>
<thead>
<tr>
<th>ErrorCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>The <code>path</code> argument names the slave side of a pseudo-terminal device that is locked.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The value of the <code>oflag</code> argument is not valid.</td>
</tr>
</tbody>
</table>
open(2)

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.

**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:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO** intro(3), 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), unlockpt(3C), attributes(5), fcntl(3HEAD), lf64(5), stat(3HEAD), 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.
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:

```
<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>
```

SEE ALSO

`alarm(2)`, `kill(2)`, `wait(2)`, `signal(3C)`, `attributes(5)`
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.

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.

The `pcsample()` function will fail if:

- `EINVAL` The value of `nsamples` is not valid.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
<tr>
<td>Interface Stability</td>
<td>Stable</td>
</tr>
</tbody>
</table>

See also

- `exec(2)`, `fork(2)`, `profil(2)`, `sysconf(3C)`, `attributes(5)`
#include <unistd.h>

int pipe(int fildes[2]);

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 and O_NONBLOCK flags are cleared.

A read from fildes[0] accesses the data written to fildes[1] on a first-in-first-out (FIFO) basis and a read from fildes[1] accesses the data written to fildes[0] also on a FIFO basis.

The FD_CLOEXEC flag will be clear on both file descriptors.

Upon successful completion pipe() marks for update the st_atime, st_ctime, and st_mtime fields of the pipe.

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

The pipe() function will fail if:

* ENFILE A file table entry could not be allocated.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO sh(1), fcntl(2), fstat(2), getmsg(2), poll(2), putmsg(2), read(2), write(2), attributes(5), streamio(7)

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).
#include <poll.h>

int poll(struct pollfd fds[], nfds_t nfds, int timeout);

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:

```c
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.
poll(2)

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, POLLRDNDM, 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 zero, events is ignored and revents is set to 0 in that entry on return from poll().

The results of the poll() query are stored in the revents 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 INFTIM (or −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:
### poll(2)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>Allocation of internal data structures failed, but the request may be attempted again.</td>
</tr>
<tr>
<td>EFAULT</td>
<td>Some argument points to an illegal address.</td>
</tr>
<tr>
<td>EINTR</td>
<td>A signal was caught during the <code>poll()</code> function.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The argument <code>nfds</code> is greater than <code>{OPEN_MAX}</code>, or one of the <code>fd</code> members refers to a STREAM or multiplexer that is linked (directly or indirectly) downstream from a multiplexer.</td>
</tr>
</tbody>
</table>

### SEE ALSO
- intro(3), getmsg(2), getrlimit(2), putmsg(2), read(2), write(2), select(3C), chpoll(9E)
- STREAMS Programming Guide

### NOTES
Non-STREAMS drivers use `chpoll(9E)` to implement `poll()` on these devices.
#include <sys/types.h>
#include <sys/processor.h>

int p_online(processorid_t processorid, int flag);

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`, and `P_NOINTR`.

When `flag` is `P_STATUS`, no processor status change occurs, but the current processor status is returned.

The `P_ONLINE`, `P_OFFLINE`, and `P_NOINTR` values for `flag` refer to valid processor states. 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. At least one processor must always be available to service system clock interrupts.

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 number of processors present can be determined by calling `sysconf(_SC_NPROCESSORS_CONF)`. The list of valid processor numbers can be determined by calling `p_online()` with `processorid` values starting at 0 until all processors have been found. The `EINVAL` error is returned for invalid processor numbers. See EXAMPLES below.

On successful completion, the value returned is the previous state of the processor, `P_ONLINE`, `P_OFFLINE`, `P_NOINTR`, or `P_POWEROFF`. Otherwise, −1 is returned and `errno` is set to indicate the error.

The `p_online()` function will fail if:

- `EPERM` The effective user of the calling process is not super-user.
- `EINVAL` A non-existent processor ID was specified or `flag` was invalid.
 EBUSY  The flag was P_OFFLINE 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.

 EBUSY  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.

 EBUSY  The specified processor is powered off and cannot be powered on because some platform-specific resource is not available.

 ENOTSUP  The specified processor is powered off, and the platform does not support power on of individual processors.

EXAMPLE 1  List the legal processor numbers.

The following code sample will list the legal processor numbers:

```c
#include <sys/unistd.h>
#include <sys/processor.h>
#include <sys/types.h>
#include <stdio.h>
#include <errno.h>

int main()
{
    processorid_t i;
    int status;
    int n = sysconf(_SC_NPROCESSORS_ONLN);
    for (i = 0; n > 0; i++) {
        status = p_online(i, P_STATUS);
        if (status == -1 && errno == EINVAL)
            continue;
        printf("processor %d present\n", i);
        n--;
    }
    return (0);
}
```

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  psadm(1M), psrinfo(1M), processor_bind(2), processor_info(2), pset_create(2), sysconf(3C), attributes(5)
priocntl – process scheduler control

SYNOPSIS
#include <sys/types.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>

long priocntl(idtype_t idtype, id_t id, int cmd, /* arg */ ...);

DESCRIPTION
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 two classes currently supported are the realtime class and the time-sharing class. The characteristics of these classes are described under the corresponding headings below. The class attribute of an LWP is inherited across the fork(2) and _lwp_create(2) functions and the exec family of functions (see exec(2)). 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 is to apply. 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:

<table>
<thead>
<tr>
<th>idtype</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_LWPID</td>
<td>The id argument is an LWP ID. The priocntl function applies to the LWP with the specified ID within the calling process.</td>
</tr>
<tr>
<td>P_PID</td>
<td>The id argument is a process ID specifying a single process. The priocntl() function applies to all LWPs currently associated with the specified process.</td>
</tr>
<tr>
<td>P_PPID</td>
<td>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.</td>
</tr>
<tr>
<td>P_PGID</td>
<td>The id argument is a process group ID. The priocntl() function applies to all LWPs currently associated with processes in the specified process group.</td>
</tr>
<tr>
<td>P_SID</td>
<td>The id argument is a session ID. The priocntl() function applies to all LWPs currently associated with processes in the specified session.</td>
</tr>
</tbody>
</table>
The id argument is a task ID. The `priocntl()` function applies to all LWPs currently associated with processes in the specified task.

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.

The id argument is a user ID. The `priocntl()` function applies to all LWPs with this effective user ID.

The id argument is a group ID. The `priocntl()` function applies to all LWPs with this effective group ID.

The id argument is a project ID. The `priocntl()` function applies to all LWPs with this project ID.

The `priocntl()` function applies to all existing LWPs. The value of `id` is ignored. The permission restrictions described below still apply.

An `id` value of `P_MYID` can be used in conjunction with the `idtype` value to specify the calling LWP’s LWP ID, parent process ID, process group ID, session ID, task ID, class ID, user ID, group ID, or project ID.

In order to change the scheduling parameters of an LWP (using the `PC_SETPARMS` command as explained below) the real or effective user ID of the LWP calling `priocntl()` must match the real or effective user ID of the receiving LWP or the effective user ID of the calling LWP must be super-user. These are the minimum permission requirements enforced for all classes. An individual class may impose additional permissions requirements when setting LWPs to that class and/or when setting class-specific scheduling parameters.

A special `sys` scheduling class exists for the purpose of scheduling the execution of certain special system processes (such as the swapper process). It is not possible to change the class of any LWP to `sys`. In addition, any processes in the `sys` class that are included in a specified set of processes are disregarded by `priocntl()`. For example, an `idtype` of `P_UID` and an `id` value of 0 would specify all processes with a user ID of 0 except processes in the `sys` class and (if changing the parameters using `PC_SETPARMS`) the `init(1M)` process.

The `init` process is a special case. In order for a `priocntl()` call to change the class or other scheduling parameters of the `init` process (process ID 1), it must be the only process specified by `idtype` and `id`. The `init` process may be assigned to any class configured on the system, but the time-sharing class is almost always the appropriate choice. (Other choices may be highly undesirable; see the `System Administration Guide, Volume 1` for more information.)

The data type and value of `arg` are specific to the type of command specified by `cmd`.

A structure with the following members is used by the `PC_GETCID` and `PC_GETCLINFO` commands.
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 or TS for time-sharing).

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 or TIME-SHARING CLASS) below.

A structure with the following elements is used by the `PC_SETPARMS` and `PC_GETPARMS` commands.

```c
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>`.

**COMMANDS**

Available `priocntl()` commands are:

**PC_GETCID**

Get class ID and class attributes for a specific class given class name. The `idtype` and `id` arguments are ignored. If `arg` is non-null, it points to a structure of type `pcinfo_t`. The `pc_clname` buffer contains the name of the class whose attributes you are getting.

On success, the class ID is returned in `pc_cid`, the class attributes are returned in the `pc_clinfo` buffer, and the `priocntl()` call returns the total number of classes configured in the system (including the `sys` class). If the class specified by `pc_clname` is invalid or is not currently configured the `priocntl()` call returns `-1` with `errno` set to `EINVAL`. The format of the attribute data returned for a given class is defined in the `<sys/rtpriocntl.h>` or `<sys/tspriocntl.h>` header 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> or <sys/tspriocntl.h> header file and described under the appropriate heading below.

If arg is a null pointer, no attribute data is returned but the priocntl() call still returns the number of configured classes.

PC_SETPARMS

Set the class and class-specific scheduling parameters of the specified LWP(s) associated with the specified process(es). When this command is used with the idtype of P_LWPID, it will set the class and class-specific scheduling parameters of the LWP. The arg member 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> or <sys/tspriocntl.h> header and described under the appropriate class heading below.

When setting parameters for a set of LWPs, priocntl() acts on the LWPs in the set in an implementation-specific order. If priocntl() encounters an error for one or more of the target processes, it may or may not continue through the set of LWPs, depending on the nature of the error. If the error is related to permissions (EPERM), priocntl() continues through the LWP set, resetting the parameters for all target LWPs for which the calling LWP has appropriate permissions. 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_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_ADMIN**

This command provides functionality needed for the implementation of the `dispadmin(1M)` utility. It is not intended for general use by other applications.

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 may be assigned to an LWP within the class. Real-time priorities range from 0 to `x`, where the value of `x` is configurable and can be determined for a specific installation by using the `priocntl()` `PC_GETCID` or `PC_GETCLINFO` command.

The 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 may run assuming that it does not complete or enter a resource or event wait state (`sleep`). Note that if another LWP becomes runnable at a higher priority, the currently running LWP may be preempted before receiving its full time quantum.

The system’s process scheduler keeps the runnable 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). Real-time LWPs that wake up after sleeping, LWPs which change to the realtime class from some other class, LWPs which have used their full time quantum, and runnable
LWs whose priority is reset by priocntl() are all placed at the back of the appropriate queue for their priority. An LWP that is preempted by a higher priority LWP remains at the front of the queue (with whatever time is remaining in its time quantum) and runs before any other LWP at this priority. Following a fork(2) or _lwp_create(2) 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 structure with the following members (defined in <sys/rtpriocntl.h>) defines the format used for the attribute data for the realtime class.

```c
class 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 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.

```c
class rt_pri; /* Real-Time priority */
class rt_tqsecs; /* Seconds in time quantum */
class 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 is in this format.

The above 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. Note that a running LWP can voluntarily release the CPU and go to the back of the scheduling queue at the same priority by resetting its rt_pri value to its current realtime priority value. In order to change the time quantum of an LWP without setting the priority or affecting the LWP’s position on the queue, the rt_pri 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_qsecs` to 2 and `rt_qnsecs` 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_qnsecs` 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 `LONG_MAX` ticks (defined in `<limits.h>`). Requesting a quantum greater than this maximum results in an error return with `errno` set to `ERANGE` (although infinite quantums may be requested using a special value as explained below). Requesting a time quantum of 0 (setting both `rt_qsecs` and `rt_qnsecs` to 0) results in an error return with `errno` set to `EINVAL`.

The `rt_qnsecs` member can also be set to one of the following special values (defined in `<sys/rtprioctl.h>`), in which case the value of `rt_qsecs` 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`.

In order to change the class of an LWP to realtime (from any other class) the LWP invoking `priocntl()` must have super-user privileges. In order to change the priority or time quantum setting of a realtime LWP, the LWP invoking `priocntl()` must have super-user 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 (see `exec(2)`).

**TIME-SHARING CLASS**

The time-sharing scheduling policy provides for a fair and effective allocation of the CPU resource among LWPs with varying CPU consumption characteristics. The objectives of the time-sharing policy are to provide good response time to interactive LWPs and good throughput to CPU-bound jobs while providing a degree of user/application control over scheduling.

The time-sharing class has a range of time-sharing user priority (see `ts_upri` below) values that may be assigned to LWPs within the class. A `ts_upri` value of 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. This is because the ts_upri value is just one factor used to determine the scheduling priority of a time-sharing LWP. The system may dynamically adjust the internal scheduling priority of a time-sharing LWP based on other factors such as recent CPU usage.

In addition to the system-wide limits on user priority (returned by the PC_GETCID and PC_GETCLINFO commands) there is a per LWP user priority limit (see ts_uprilim below), which specifies the maximum ts_upri value that may be set for a given LWP; by default, ts_uprilim is 0.

A structure with the following members (defined in <sys/tspriocntl.h>) defines the format used for the attribute data for the time-sharing class.

```c
short ts_maxupri; /* Limits of user priority range */
```

The priocntl() PC_GETCID and PC_GETCLINFO commands return time-sharing class attributes in the pc_clinfo buffer in this format.

ts_maxupri specifies the configured maximum user priority value for the time-sharing class. If ts_maxupri is x, the valid range for both user priorities and user priority limits is from -x to +x.

A structure with the following members (defined in <sys/tspriocntl.h>) defines the format used to specify the time-sharing class-specific scheduling parameters of an LWP.

```c
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_SETPARMS command, if pc_cid specifies the time-sharing class and more than one time-sharing LWP is specified, the scheduling parameters of the time-sharing LWP with the highest ts_upri value among the specified LWPs is returned and the LWP ID of this LWP is returned by the priocntl() call. If there is more than one LWP sharing the highest user priority, the one returned is implementation-dependent.

Any time-sharing LWP may lower its own ts_uprilim (or that of another LWP with the same user ID). Only a time-sharing LWP with super-user privileges may raise a ts_uprilim. When changing the class of an LWP to time-sharing from some other class, super-user privileges are required in order to set the initial ts_uprilim to a value greater than 0. Attempts by a non-super-user 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 may set its own `ts_upri` (or that of another LWP with the same user ID) to any value less than or equal to the LWP’s `ts_uprilim`. Attempts to set the `ts_upri` above the `ts_uprilim` (and/or set the `ts_uprilim` below the `ts_upri`) result in the `ts_upri` being set equal to the `ts_uprilim`.

Either of the `ts_uprilim` or `ts_upri` members may be set to the special value `TS_NOCHANGE` (defined in `<sys/tspriocntl.h>`) in order to set one of the values without affecting the other. Specifying `TS_NOCHANGE` for the `ts_upri` when the `ts_uprilim` is being set to a value below the current `ts_upri` causes the `ts_upri` to be set equal to the `ts_uprilim` being set. Specifying `TS_NOCHANGE` for a parameter when changing the class of an LWP to time-sharing (from some other class) causes the parameter to be set to a default value. The default value for the `ts_uprilim` is 0 and the default for the `ts_upri` is to set it equal to the `ts_uprilim` which is being set.

The time-sharing user priority and user priority limit are inherited across `fork()` and the `exec` family of functions.

**RETURN VALUES**

Unless otherwise noted above, `priocntl()` returns a value of 0 on success. On failure, `priocntl()` returns −1 and sets `errno` to indicate the error.

**ERRORS**

The `priocntl()` function fails 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 effective user of the calling LWP is not super-user.

- **ERANGE**: The requested time quantum is out of range.

- **ESRCH**: None of the specified LWPs exist.

**SEE ALSO**

`priocntl(1), disadmin(1M), init(1M), _lwp_create(2), exec(2), fork(2), nice(2), priocntlset(2), rt_dptbl(4)`

*System Administration Guide, Volume 1*

*System Interface Guide*
#include <sys/types.h>
#include <sys/procset.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>

long priocntlset(procset_t *psp, int cmd, /* arg */ ...);

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),
Unless otherwise noted above, `priocntlset()` returns 0 on success. Otherwise, it returns -1 and sets `errno` to indicate the error.

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

- **ERANGE**: The requested time quantum is out of range.

- **ESRCH**: None of the specified processes exist.

**SEE ALSO**

`priocntl(1), priocntl(2)`
**NAME**
processor_bind – bind LWPs to a processor

**SYNOPSIS**
```c
#include <sys/types.h>
#include <sys/processor.h>
#include <sys/proceset.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 `id` is `P_MYID`, the specified LWP, process, or task is the current one.

If `processorid` is `PBIND_NONE`, the processor bindings of the specified LWPs are cleared.

If `processorid` is `PBIND_QUERY`, the processor bindings are not changed.

The effective user of the calling process must be superuser, or its real or effective user ID must match the real or effective user ID of the LWPs being bound. If the calling process does not have permission to change all of the specified LWPs, the bindings of the LWPs for which it does have permission will be changed even though an error is returned.

**RETURN VALUES**
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`, or `P_TASKID`.
- **EPERM**: The effective user of the calling process is not superuser, 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`.

---

processor_bind(2)

**NAME**
processor_bind – bind LWPs to a processor

**SYNOPSIS**
```c
#include <sys/types.h>
#include <sys/processor.h>
#include <sys/proceset.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 `id` is `P_MYID`, the specified LWP, process, or task is the current one.

If `processorid` is `PBIND_NONE`, the processor bindings of the specified LWPs are cleared.

If `processorid` is `PBIND_QUERY`, the processor bindings are not changed.

The effective user of the calling process must be superuser, or its real or effective user ID must match the real or effective user ID of the LWPs being bound. If the calling process does not have permission to change all of the specified LWPs, the bindings of the LWPs for which it does have permission will be changed even though an error is returned.

**RETURN VALUES**
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`, or `P_TASKID`.
- **EPERM**: The effective user of the calling process is not superuser, 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`. 
processor_bind(2)

SEE ALSO: psadm(1M), psinfo(1M), p_online(2), pset_bind(2), sysconf(3C)
processor_info(2)

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, 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.
EFAULT  The processor_info_t structure pointed to by infop was not
writable by the user.

SEE ALSO
psradm(1M), psrinfo(1M), p_online(2), sysconf(3C)
NAME
profil – execution time profile

SYNOPSIS
#include <unistd.h>

void profil(unsigned short *buff, unsigned int bufsiz, unsigned int offset, unsigned int scale);

DESCRIPTION
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.

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 \((RATIO \times 0.00000200L)\), where \(RATIO\) is the desired ratio of \(bufsz\) 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 \(bufsz\) can be computed as \((size\_of\_region\_to\_be\_profiled \times RATIO)\).

Profiling is turned off by giving a \(scale\) value of 0 or 1, and is rendered ineffective by giving a \(bufsz\) 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 obind 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 id is P_MYID, the specified LWP or process 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.

The effective user of the calling process must be super-user, or its real or effective user ID must match the real or effective user ID of the LWPs being bound, or pset must be PS_QUERY. 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.

If the processor set type of pset is PS_PRIVATE (see pset_info(2)), the effective user of the calling process must be super-user.

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.

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, or P_TASKID.
The effective user of the calling process is not super-user, and either the processor set type of `pset` is `PS_USER`, or 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.

No processes, LWPs, or tasks were found to match the criteria specified by `idtype` and `id`.

**SEE ALSO**

`pbind(1M), psrset(1M), processor_bind(2), pset_create(2), pset_info(2)`
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.

Only a limited number of processor sets may be active (created and not destroyed) at a given time. This limit will always be greater than the number of processors in the system. If pset_create() is called when the maximum number of processor sets is already active, the function will return -1 and errno will be set to ENOMEM.

The pset_destroy() function destroys the processor set pset, releasing its constituent processors and processes.

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.

These functions are restricted to super-user use, 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.
EPERM   The effective user of the calling process is not super-user.

SEE ALSO  psradm(1M), psrinfo(1M), psrset(1M), p_online(2), processor_bind(2), pset_bind(2), pset_info(2)

NOTES  Processors belonging to different processor sets of type PS_SYSTEM (see pset_info(2)) cannot be assigned to the same processor set of type PS_PRIVATE. If this is attempted, pset_assign() will fail and set errno to EINVAL.

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.
pset_info(2)

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. Processor set types can have the following values:

- **PS_SYSTEM**: The processor set was created by the system. Processor sets of this type cannot be modified or removed by the user, but LWPs and processes can be bound to them using pset_bind(2).

- **PS_PRIVATE**: The processor set was created by pset_create(2) and can be modified by pset_assign(2) and removed by pset_destroy(2). LWPs and processes can also be bound to this processor set using pset_bind().

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.

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.

SEE ALSO
psrinfo(1M), pscatter(1M), processor_info(2), pset_assign(2), pset_bind(2), pset_create(2), pset_destroy(2)
### NAME
ptrace – allows a parent process to control the execution of a child process

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

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

### DESCRIPTION
The `ptrace()` function allows a parent process to control the execution of a child process. Its primary use is for the implementation of breakpoint debugging. The child process behaves normally until it encounters a signal (see `signal(3HEAD)`), at which time it enters a stopped state and its parent is notified via the `wait(2)` function. When the child is in the stopped state, its parent can examine and modify its “core image” using `ptrace()`. Also, the parent can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

The `request` argument determines the action to be taken by `ptrace()` and is one of the following:

- **0**
  This request must be issued by the child process if it is to be traced by its parent. It turns on the child’s trace flag that stipulates that the child should be left in a stopped state on receipt of a signal rather than the state specified by `func` (see `signal(3C)`). The `pid`, `addr`, and `data` arguments are ignored, and a return value is not defined for this request. Peculiar results ensue if the parent does not expect to trace the child.

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

- **1, 2**
  With these requests, the word at location `addr` in the address space of the child is returned to the parent process. If instruction and data space are separated, request 1 returns a word from instruction space, and request 2 returns a word from data space. If instruction and data space are not separated, either request 1 or request 2 may be used with equal results. The `data` argument is ignored. These two requests fail if `addr` is not the start address of a word, in which case −1 is returned to the parent process and the parent’s `errno` is set to EIO.

- **3**
  With this request, the word at location `addr` in the child’s user area in the system’s address space (see `<sys/user.h>`) is returned to the parent process. The `data` argument is ignored. This request fails if `addr` is not the start address of a word or is outside the user area, in which case −1 is returned to the parent process and the parent’s `errno` is set to EIO.

- **4, 5**
  With these requests, the value given by the `data` argument is written into the address space of the child at location `addr`. If instruction and data space are separated, request 4 writes a word into instruction space, and request 5 writes a word into data space. If instruction and data space are not separated, either request 4 or request 5 may be used with equal results. On success, the value written into the address space of the child is returned to...
the parent. These two requests fail if addr is not the start address of a word. On failure −1 is returned to the parent process and the parent’s errno is set to EIO.

With this request, a few entries in the child’s user area can be written. data gives the value that is to be written and addr is the location of the entry. The few entries that can be written are the general registers and the condition codes of the Processor Status Word.

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

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

This request sets the trace bit in the Processor Status Word of the child and then executes the same steps as listed above for request 7. The trace bit causes an interrupt on completion of one machine instruction. This effectively allows single stepping of the child.

To forestall possible fraud, ptrace() inhibits the set-user-ID facility on subsequent calls to one of the exec family of functions (see exec(2)). If a traced process calls one of the exec functions, it stops before executing the first instruction of the new image showing signal SIGTRAP.

ERRORS

The ptrace() function will fail if:

- **EIO** The request argument is an illegal number.
- **EPERM** The effective user of the calling process is not super-user.
- **ESRCH** The pid argument identifies a child that does not exist or has not executed a ptrace() call with request 0.

SEE ALSO exec(2), exit(2), wait(2), signal(3C), signal(3HEAD)
NAME
putmsg, putpmsg – send a message on a stream

SYNOPSIS
#include <stropts.h>

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

int putpmsg(int fd, 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 fd 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 fieldes argument is not a valid file descriptor open for writing.
-EFAULT: The cctlptr 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 fieldes 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 fieldes 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 fieldes argument refers to a STREAMS-based pipe and the other end of the pipe is closed. A SIGPIPE signal is generated for the calling process. This error condition occurs only with SUS-compliant applications. See standards(5).
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.

**SEE ALSO**

- `intro(2)`, `getmsg(2)`, `poll(2)`, `read(2)`, `write(2)`, `standards(5)`

*STREAMS Programming Guide*
The `read()` function attempts to read `nbyte` bytes from the file associated with the open file descriptor, `fd`, into the buffer pointed to by `buf`.

If `nbyte` is 0, `read()` will return 0 and have 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 `fd`. 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 `fd` 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.

If the value of `nbyte` is greater than `SSIZE_MAX`, the result 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`.
read(2)

- 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.

The `readv()` function is equivalent to `read()`, but places the input data into the `iovcnt` buffers specified by the members of the `iov` array: `iov0`, `iov1`, ... , `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:

```c
typedef struct {
  caddr_t  iov_base;
  int      iov_len;
} iovec;
```

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 `filedes` 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.
- **EFAULT** The `buf` argument points to an illegal address.
- **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.
- **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 `filedes` argument is on a remote machine and the link to that machine is no longer active.
The device associated with `fd` is a block special or character special file and the value of the file pointer is out of range.

The `read()` and `readv()` functions will fail if:

- **ENOERR** 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 `fd`.

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 or equal to `{IOV_MAX}`. (See `intro(3)` for a definition of `{IOV_MAX}`).
- **EINVAL** The sum of the `iov_len` values in the `iov` array overflowed an int.

The `pread()` function will fail and the file pointer remain unchanged if:

- **ESPIPE** The `fd` argument is associated with a pipe or FIFO.

**USAGE**

The `pread()` function has a transitional interface for 64-bit file offsets. See `l64(5)`.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td><code>read()</code> is Async-Signal-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`intro(3)`, `chmod(2)`, `creat(2)`, `dup(2)`, `fcntl(2)`, `getmsg(2)`, `ioctl(2)`, `lseek(2)`, `open(2)`, `pipe(2)`, `recv(3SOCKET)`, `attributes(5)`, `l64(5)`, `streamio(7I)`, `termio(7I)`
readlink (2)

NAME    readlink – read the contents of a symbolic link

SYNOPSIS #include <unistd.h>

    int readlink(const char *path, char *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 unspecified.

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.

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

ELOOP    Too many symbolic links were encountered in resolving path.

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.

ENAMETOOLONG Path name resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.

USAGE Portable applications should not assume that the returned contents of the symbolic link are null-terminated.

SEE ALSO stat(2), symlink(2)
rename – change the name of a file

#include <stdio.h>

int rename(const char *old, const char *new);

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 pathname of the file.

If old and new both refer to the same existing file, the rename() function returns successfully and performs no other action.

If old points to the pathname of a file that is not a directory, new must not point to the pathname of a directory. If the link named by new exists, it will be removed and old will be renamed to new. In this case, a link named new must remain visible to other processes throughout the renaming operation and will refer to either the file referred to by new or the file referred to as old before the operation began.

If old points to the pathname of a directory, new must not point to the pathname of a file that is not a directory. If the directory named by new exists, it will be removed and old will be renamed to new. In this case, a link named new will exist throughout the renaming operation and will refer to either the file referred to by new or the file referred to as old before the operation began. Thus, if new names an existing directory, it must be an empty directory.

The new pathname must not contain a path prefix that names old. Write access permission is required for both the directory containing old and the directory containing new. If old points to the pathname of a directory, write access permission is required for the directory named by old, and, if it exists, the directory named by new.

If the directory containing old has the sticky bit set, at least one of the following conditions listed below must be true:

- the user must own old
- the user must own the directory containing old
- old must be writable by the user
- the user must be a privileged user

If new exists, and the directory containing new is writable and has the sticky bit set, at least one of the following conditions must be true:

- the user must own new
- the user must own the directory containing new
- new must be writable by the user
- the user must be a privileged user
rename(2)

If the link named by new exists, the file’s link count becomes zero when it is removed, and no process has the file open, then the space occupied by the file will be freed and the file will no longer be accessible. If one or more processes have the file open when the last link is removed, the link will be removed before rename() returns, but the removal of the file contents will be postponed until all references to the file have been closed.

Upon successful completion, the rename() function will mark for update the st_ctime and st_mtime fields of the parent directory of each file.

RETURN VALUES
Upon successful completion, 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).

EINVAL The new argument directory pathname contains a path prefix that names the old directory.

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 exist, or either old or new points to an empty string.

ENOSPC The directory that would contain new cannot be extended.
ENOTDIR A component of either path prefix is not a directory, or old names a directory and new names a nondirectory file.

EROFS The requested operation requires writing in a directory on a read-only file system.

EXDEV The links named by old and new are on different file systems.

EIO An I/O error occurred while making or updating a directory entry.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO chmod(2), link(2), unlink(2), attributes(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.
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.

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.

The `resolvepath()` function will fail if:

- **EACCESS** 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.

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)`
rmdir – remove a directory

#include <unistd.h>

int rmdir(const char *path);

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.

Upon successful completion, 0 is returned. Otherwise, −1 is returned, `errno` is set to indicate the error, and the named directory is not changed.

The `rmdir()` function will fail if:

- `EACCES` Search permission is denied for a component of the path prefix; write permission is denied on the directory containing the directory to be removed; the parent directory has the `S_ISVTX` variable set and is not owned by the user; the directory is not owned by the user and is not writable by the user; or the user is not a super-user.
- `EBUSY` The directory to be removed is the mount point for a mounted file system.
- `EEXIST` The directory contains entries other than those for "." and "..".
- `EFAULT` 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 ".".
- `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.
rmmdir(2)

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:

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

See also  mkdir(1), rm(1), mkdir(2), attributes(5)
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.
semctl(2)

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 an effective user ID equal to either that of super-user, or to the value of `sem_perm.cuid` or `sem_perm.uid` in the data structure associated with `semid`.

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

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

- **GETVAL**
  the value of `semval`
- **GETPID**
  the value of `(int) sempid`
- **GETNCNT**
  the value of `semncnt`
- **GETZCNT**
  the value of `semzcnt`

All other successful completions return 0; otherwise, −1 is returned and `errno` is set to indicate the error.

**ERRORS**
The `semctl()` function will fail if:

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

- **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` and the effective user of the calling process is not super-user, or `cmd` is equal to the value of `sem_perm.cuid` or `sem_perm.uid` in the data structure associated with `semid`.
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.

SEE ALSO  ipcs(1), intro(2), semget(2), semop(2)
semget(2)

NAME  | semget – get set of semaphores

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

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

DESCRIPTION  
The semget() function returns the semaphore identifier associated with key.
A semaphore identifier and associated data structure and set containing nsems semaphores (see intro(3)) 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(3)) 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; or 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 ALSO ipcrm(1), ipcs(1), intro(3), semctl(2), semop(2), ftok(3C)
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.

```c
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++)
```

234 man pages section 2: System Calls • Last Revised 8 Mar 2000
EXAMPLE 1 semids() example  (Continued)

    process_semid(ids[i]);
    free(ids);
}

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

ipcrm(1), ipcs(1), intro(2), semct1(2), semget(2), semop(2), attributes(5)
semop(2)

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(3) for more information.

The sem_op member specifies one of three semaphore operations:

1. The sem_op member is a negative integer; {ALTER}
   - If semval (see intro(3)) 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 & IPC_NOWAIT) is false, semop() increments the semncnt associated
     with the specified semaphore and suspends execution of the calling process
     until one of the following conditions occur:
   - If sem_op is less than the absolute value of sem_op and
     (sem_flg & IPC_NOWAIT) is true, semop() returns immediately.
   - If semval is less than the absolute value of sem_op and
     (sem_flg & IPC_NOWAIT) is false, semop() increments the semncnt associated
     with the specified semaphore and suspends execution of the calling process
     until one of the following conditions occur:
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 process 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 process receives a signal that is to be caught. When this occurs, the value of `semncnt` associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in `signal(3C)`.

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 process 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 decremented.
     - The `semid` for which the calling process is awaiting action is removed from the system. When this occurs, `errno` is set to `EIDRM` and −1 is returned.
     - The calling process receives a signal that is to be caught. When this occurs, the value of `semzcnt` associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in `signal(3C)`.

Upon successful completion, the value of `semipid` 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()`.

System Calls 237
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.

- **EACCES**  
Operation permission is denied to the calling process (see `intro(3)`).

- **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.

- **EINVAL**  
A signal was received.

- **EINTR**  
The `semid` argument is not a valid semaphore identifier, or the number of individual semaphores for which the calling process requests a `SEM_UNDO` would exceed the limit.

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

- **ERANGE**  
An operation would cause a `semval` or a `semadj` value to overflow the system-imposed limit.

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.

### SEE ALSO

*ipcs(1), intro(3), exec(2), exit(2), fork(2), semctl(2), semget(2)*)
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.

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

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.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO** `intro(2), exec(2), exit(2), fork(2), getpid(2), getsid(2), attributes(5)`
NAME  setpgroup – set process group ID

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

            pid_t setpgroup(void);

DESCRIPTION  If the calling process is not already a session leader, the setpgroup() 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(3) for more information on process group
              IDs and session leaders.

RETURN VALUES  The setpgroup() function returns the value of the new process group ID.

SEE ALSO  intro(3), exec(2), fork(2), getpid(2), getsid(2), kill(2), signal(3C)

NOTES  The setpgroup() function will be phased out in favor of the setsid(2) function.
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 effective user ID of the calling process is super-user, the real group ID and the effective group ID can be set to any legal value.

If the effective user ID of the calling process is not super-user, 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.

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.

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 calling process’s effective UID is not the super-user and a change 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, was specified.

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.
setreuid(2)

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 effective user ID of the calling process is super-user, the real user ID and the effective user ID can be set to any legal value.

If the effective user ID of the calling process is not super-user, either the real user ID can be set to the effective user ID, or the effective user ID can either be set to the saved set-user ID from execve() (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.

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 calling process’s effective user ID is not the super-user and a change other than changing the real user ID to the effective user ID, or changing the effective user ID to the real user ID or the saved set-user ID, was specified.

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.
SEE ALSO | exec(2), getuid(2), setregid(2), setuid(2)
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:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
getsid(2), setpgid(2), setpgrp(2), attributes(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.
settaskid(2)

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);
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 superuser 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.
EPERM   The effective user of the calling process is not superuser.
EINVAL  The given project ID is not within the valid project ID range.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
setsid(2), project(4), attributes(5)
setuid(), setegid(), seteuid(), setgid() – set user and group IDs

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

int setuid(uid_t uid);
int setegid(gid_t egid);
int seteuid(uid_t euid);
int setgid(gid_t gid);

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 family of functions (see exec(2)) to execute a file
(program), the user and/or group identifiers associated with the process can change. If
the file executed is a set-user-ID file, the effective and saved user IDs of the process are
set to the owner of the file executed. If the file executed is a set-group-ID file, the
effective and saved group IDs of the process are set to the group of the file executed. If
the file executed is not a set-user-ID or set-group-ID file, the effective user ID, saved
user ID, effective group ID, and saved group ID are not changed.

If the effective user ID of the process calling setuid() is the super-user, the real,
effective, and saved user IDs are set to the uid argument.

If the effective user ID of the calling process is not the super-user, but uid is either the
real user ID or the saved user ID of the calling process, the effective user ID is set to
uid.

If the effective user ID of the process calling setgid() is the super-user, the real,
effective, and saved group IDs are set to the gid argument.

If the effective user ID of the calling process is not the superuser, 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 effective user of the calling process is not super-user, and the *uid* argument does not match either the real or saved user IDs. For `setgid()` and `setegid()` the effective user of the calling process is not the superuser, and the *gid* argument does not match either the real or saved group IDs.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td><code>setuid()</code> and <code>setgid()</code> and</td>
</tr>
<tr>
<td></td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`intro(2), exec(2), getgroups(2), getuid(2), stat(3HEAD)` attributes(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:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IPC_STAT</strong></td>
<td>Place the current value of each member of the data structure associated with <code>shmid</code> into the structure pointed to by <code>buf</code>. The contents of this structure are defined in <code>intro(2)</code>. [READ]</td>
</tr>
</tbody>
</table>
| **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 an effective user ID equal to that of super-user, or to the value of `shm_perm.cuid` or `shm_perm.uid` in the data structure associated with `shmid`.

| **IPC_RMID** | Remove the shared memory identifier specified by `shmid` from the system and destroy the shared memory segment and data structure associated with it. This command can be executed only by a process that has an effective user ID equal to that of super-user, or to the value of `shm_perm.cuid` or `shm_perm.uid` in the data structure associated with `shmid`. |
| **SHM_LOCK** | Lock the shared memory segment specified by `shmid` in memory. This command can be executed only by a process that has an effective user ID equal to super-user. |
| **SHM_UNLOCK** | Unlock the shared memory segment specified by `shmid`. This command can be executed only by a process that has an effective user ID equal to super-user. |
**shmctl(2)**

Shared memory segments must be explicitly removed after the last reference to them has been removed.

**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.
- **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** and the effective user ID of the calling process is not super-user and it is not equal to the value of **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with **shmid**.
- **EPERM** The **cmd** argument is equal to **SHM_LOCK** or **SHM_UNLOCK** and the effective user ID of the calling process is not equal to that of super-user.

**SEE ALSO**

ipcs(1), intro(2), shmget(2), shmp(2)
shmget – get shared memory segment identifier

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

int shmget(key_t key, size_t size, int shmdflg);

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(3)) 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 (shmdflg & 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 shmdflg. shm_segsz is set equal to the value of size.
- The values of shm_lpid, shm_nattchshm_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:

EACCESS A shared memory identifier exists for key but operation permission (see intro(3)) as specified by the low-order 9 bits of shmdflg would not be granted.
EEXIST A shared memory identifier exists for key but both (shmdflg & IPC_CREAT) and (shmdflg & IPC_EXCL) are true.
EINVAL The size argument is less than the system-imposed minimum or greater than the system-imposed maximum.
EINVAL A shared memory identifier exists for key but the size of the segment associated with it is less than size and size is not equal to 0.
**shmget(2)**

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOENT</td>
<td>A shared memory identifier does not exist for <strong>key</strong> and <code>shmflg&amp;IPC_CREATE</code> is false.</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>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.</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>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.</td>
</tr>
</tbody>
</table>

**SEE ALSO**

intro(3), shmctl(2), shmap(2), ftok(3C)
shmids – discover all shared memory identifiers

#include <sys/shm.h>

int shmids(int *buf, uint_t nids, uint_t *pnids);

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.

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

The shmids() function will fail if:

EFAULT The buf or pnids argument points to an illegal address.

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.

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 */
        ids = realloc(ids, (nids = n) * sizeof (int));
    }

    for (i = 0; i < n; i++)
```

System Calls 251
EXAMPLE 1  shmids() example  (Continued)

    process_shmid(ids[i]);
    free(ids);

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  ipcrm(1), ipcs(1), intro(2), shmtel(2), shmget(2), shmp(2), attributes(5)
The \texttt{shmat()} function attaches the shared memory segment associated with the shared memory identifier specified by \texttt{shmid} to the data segment of the calling process.

The permission required for a shared memory control operation is given as \{\textit{token}\}, where \textit{token} is the type of permission needed. The types of permission are interpreted as follows:

- \texttt{00400 READ by user}
- \texttt{00200 WRITE by user}
- \texttt{00040 READ by group}
- \texttt{00020 WRITE by group}
- \texttt{00004 READ by others}
- \texttt{00002 WRITE by others}

See the \textit{Shared Memory Operation Permissions} section of \texttt{intro(2)} for more information.

When \((\texttt{shmflg}\&\texttt{SHM\_SHARE\_MMU})\) is true, virtual memory resources in addition to shared memory itself are shared among processes that use the same shared memory.

When \((\texttt{shmflg}\&\texttt{SHM\_PAGEABLE})\) 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 \texttt{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 \texttt{shmaddr} is equal to \((\texttt{void \*}) 0\), the segment is attached to the first available address as selected by the system.
- If \texttt{shmaddr} is equal to \((\texttt{void \*}) 0\) and \((\texttt{shmflg}\&\texttt{SHM\_SHARE\_MMU})\) or \((\texttt{shmflg}\&\texttt{SHM\_PAGEABLE})\) is true, then the segment is attached to the first available suitably aligned address. When \((\texttt{shmflg}\&\texttt{SHM\_SHARE\_MMU})\) or \((\texttt{shmflg}\&\texttt{SHM\_PAGEABLE})\) is set, however, the permission given by \texttt{shmget()} determines whether the segment is attached for reading or reading and writing.
- If \texttt{shmaddr} is not equal to \((\texttt{void \*}) 0\) and \((\texttt{shmflg}\&\texttt{SHM\_RND})\) is true, the segment is attached to the address given by \((\texttt{shmaddr} - (\texttt{shmaddr} \mod \texttt{SHMLBA}))\).
- If \texttt{shmaddr} is not equal to \((\texttt{void \*}) 0\) and \((\texttt{shmflg}\&\texttt{SHM\_RND})\) is false, the segment is attached to the address given by \texttt{shmaddr}.
- The segment is attached for reading if \((\texttt{shmflg}\&\texttt{SHM\_RDONLY})\) is true \{\texttt{READ}\}, otherwise it is attached for reading and writing \{\texttt{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.
- **EINVAL** The `shmaddr` argument is not equal to 0, and the value of `(shmaddr - (shmaddr modulus SHMLBA))` is an illegal address.
- **EINVAL** The `shmaddr` argument is not equal to 0, is an illegal address, and `(shmflg & SHM_RND)` is false.
- **EINVAL** The `shmaddr` argument is not equal to 0, is not properly aligned, and `(shmflg & SHM_SHARE_MMU)` is true.
- **EINVAL** `SHM_SHARE_MMU` is not supported in certain architectures.
- **EMFILE** The number of shared memory segments attached to the calling process would exceed the system-imposed limit.
- **ENOMEM** The available data space is not large enough to accommodate the shared memory segment.

The `shmdt()` function will fail if:

- **EINVAL** The `shmaddr` argument is not the data segment start address of a shared memory segment.

**SEE ALSO**

`intro(2)`, `exec(2)`, `exit(2)`, `fork(2)`, `shmct1(2)`, `shmat(2)`, `standards(5)`
The `sigaction()` function allows the calling process to examine or specify the action to be taken on delivery of a specific signal. (See `signal(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(3HEAD)` except `SIGKILL` and `SIGSTOP`. In a multithreaded process, `sig` cannot be `SIGWAITING`, `SIGCANCEL`, or `SIGLWP`.

If the argument `act` is not `NULL`, it points to a structure specifying the new action to be taken when delivering `sig`. If the argument `oact` is not `NULL`, it points to a structure where the action previously associated with `sig` is to be stored on return from `sigaction()`.

The `sigaction` structure includes the following members:

- `void (*sa_handler)( );`
- `void (*sa_sigaction)(int, siginfo_t *, void *);`
- `sigset_t sa_mask;`
- `int sa_flags;`

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(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 LWP that is chosen to process a delivered signal has an alternate signal stack declared with `sigaltstack(2)`, then it will process the signal on that stack. Otherwise, the signal is delivered on the LWP main stack. Unbound threads (see `thr_create(3THR)`) may not have alternate signal stacks.

- **SA_RESETHAND**: If set and the signal is caught, the disposition of the signal is reset to `SIG_DFL` and the signal will not be blocked on entry to the
signal handler (SIGILL, SIGTRAP, and SIGPWR cannot be automatically reset when delivered; the system silently enforces this restriction).

**SA_NODEFER**
If set and the signal is caught, the signal will not be automatically blocked by the kernel while it is being caught.

**SA_RESTART**
If set and the signal is caught, functions that are interrupted by the execution of this signal’s handler are transparently restarted by the system, namely `fcntl(2)`, `ioctl(2)`, `wait(2)`, `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 `EINVAL` 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(3HEAD)`); the third argument points to a `ucontext_t` structure containing the receiving process’s context when the signal was delivered (see `ucontext(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(2)`, 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.

**SA_WAITSIG**
If set and `sig` equals `SIGWAITING`, enables generation of `SIGWAITING` signals. Reserved for use by the threads library.

**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`. 

---

256  man pages section 2: System Calls • Last Revised 19 Mar 1998
ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

kill(1), intro(3), exit(2), fcntl(2), getmsg(2), ioctl(2), kill(2), pause(2), putmsg(2), read(2), sigaltstack(2), sigprocmask(2), sigsend(2), sigsuspend(2), wait(2), waitid(2), write(2), recv(3SOCKET), recv(3SOCKET), signal(3C), sigsetops(3C), thr_create(3THR), attributes(5), siginfo(3HEAD), signal(3HEAD), ucontext(3HEAD)

NOTES

The handler routine can be declared:

```c
void handler (int sig, siginfo_t *sip, ucontext_t *uap);
```

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 `uap` 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 `uap` be used by the handler to restore the context from before the signal delivery.
The `sigaltstack()` function allows an LWP 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 LWP is currently executing on that stack. When a signal’s action indicates its handler should execute on the alternate signal stack (specified with a `sigaction(2)` call), the system checks whether the LWP chosen to execute the signal handler is currently executing on that stack. If the LWP is not currently executing on the signal stack, the system arranges a switch to the alternate signal stack for the duration of the signal handler’s execution.

The `stack_t` structure includes the following members:

```c
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 LWP is currently executing on the alternate signal stack. Attempts to modify the alternate signal stack while the LWP is executing on it will fail.

- **SS_DISABLE**: The alternate signal stack is currently disabled.

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

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`. 
An attempt was made to modify an active stack.

**SEE ALSO**
getcontext(2), sigaction(2), ucontext(3HEAD)

**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 sigaltstack() function creates an alternate signal stack for the calling LWP, with the implication that in a multithreaded application, only bound threads (or in POSIX terminology, only threads whose scheduling contention scope is PTHREAD_SCOPE_SYSTEM, see thr_create(3THR) and pthread_create(3THR)) should call sigaltstack(). The behavior of an application that calls sigaltstack() from an unbound thread (that is, a POSIX thread whose scheduling contention scope is PTHREAD_SCOPE_PROCESS) is undefined. In a future release, sigaltstack() may return a new error value if it is called from an unbound thread.

The following code fragment is typically used to allocate an alternate stack:

```c
if ((sigstk.ss_sp = (char *)malloc(SIGSTKSZ)) == NULL)
    /* error return */;

sigstk.ss_size = SIGSTKSZ;
sigstk.ss_flags = 0;
if (sigaltstack(&sigstk, (stack_t *)0) < 0)
    perror("sigaltstack");
```
**NAME**
_signotifywait, _lwp_sigredirect – deliver process signals to specific LWPs

**SYNOPSIS**

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

int _signotifywait(void);
int _lwp_sigredirect(lwpid_t target_lwp, int signo, int *queued);
```

**DESCRIPTION**

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

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

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

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

If there is a pending unnotified signal when _signotifywait() is called, that signal is selected and the call returns immediately. If there is not a signal pending, the call suspends the calling LWP until the generation of an unnotified signal; that signal then is selected and the function returns. In both cases, the selected signal number is set in the notification set and returned as the value of _signotifywait(). The signal remains pending for the process, and any associated siginfo(3HEAD) information remains queued at the process.

The _lwp_sigredirect() function requests that a signal pending for the process be delivered to the LWP specified by target_lwp. If target_lwp is 0, the signal is discarded. It is an error if signo is not currently in the notification set of the process. The signal specified by signo is removed from pending for the process and is made pending for the target_lwp. If there is an associated siginfo information structure queued at the process, that siginfo is queued to the target_lwp.

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

When `queued` is non-NULL, and there is another instance of the signal (`signo`), a non-zero value will be place in `queued` to indicate that more than one instance of the signal is pending on the process.

**RETURN VALUES**

The `_signotifywait()` function returns the signal number of the pending but hitherto unnotified signal. The `_lwp_sigredirect()` function returns 0 when successful. A non-zero value indicates an error.

**ERRORS**

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

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

- **EINVAL** The signal `signo` was not pending for the process, or `signo` was not in the notification set.
- **ESRCH** The target_lwp cannot be found in the current process.

**SEE ALSO**

`_lwp_create(2), _lwp_kill(2), sigtimedwait(3RT), siginfo(3HEAD), signal(3HEAD)`

**NOTES**

This mechanism for delivering signals to multithreaded processes is subject to change in future versions of Solaris. Any process with explicit knowledge of this mechanism may not be compatible from release to release.
sigpending(2)

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          |
+----------------+---------------------------+
| MT-Level       | Async-Signal-Safe        |
+----------------+---------------------------+

SEE ALSO sigaction(2), sigprocmask(2), sigsetops(3C), attributes(5)
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 there are any pending unblocked signals after the call to `sigprocmask()`, at least one of those signals will be delivered before the call to `sigprocmask()` returns.

It is not possible to block those signals that cannot be ignored this restriction is silently imposed by the system. See `sigaction(2)`.

If `sigprocmask()` fails, the caller’s signal mask is not changed.

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

The `sigprocmask()` function will fail if:

- **EFAULT**: The `set` or `oset` argument points to an illegal address.
- **EINVAL**: The value of the `how` argument is not equal to one of the defined values.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

In a multithreaded program, the call to `sigprocmask()` impacts only the calling thread’s signal mask and is therefore identical to a call to `thr_sigsetmask(3THR)`.

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**

```c
#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(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 effective user ID of the sending process is super-user, 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_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:

```c
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 is not superuser and its real or effective user ID does not match the real or effective user ID of the receiving process, and the calling process is not sending `SIGCONT` to a process that shares the same session.
- **ESRCH**: No process can be found corresponding to that specified by `id` and `idtype`.

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(3HEAD)"
sigsuspend

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 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(). See NOTES for the precise semantics of signal mask restoration in a multithreaded process.

It is not possible to block those signals that cannot be ignored (see signal(3HEAD)); this restriction is silently imposed by the system.

RETURN VALUES
Since sigsuspend() suspends process execution indefinitely, there is no successful completion return value. On failure, it returns −1 and sets errno to indicate the error.

ERRORS
The sigsuspend() function will fail if:

EFAULT The set argument points to an illegal address.
EINTR A signal was caught by the calling process and control was returned from the signal catching function.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
sigaction(2), sigprocmask(2), sigwait(2), signal(3C), sigsetops(3C), attributes (5), signal(3HEAD)

NOTES
In a multithreaded application, the sigwait(2), function should be used instead of sigsuspend(). Should sigsuspend() be used, however, its semantics of signal mask restoration are slightly different from those for a single-threaded process on return from the signal catching function, the signal mask is restored to the set that existed before the call to sigsuspend(). This action raises the following implications:

- If a thread specifies two signals in the mask to sigsuspend(), both signals could interrupt its call to sigsuspend() simultaneously. In the traditional program that does not use threads, a call to sigsuspend() with two signals in the mask always returns with only one signal delivered. The other signal remains pending if masked earlier, unlike the MT case.
While a thread is executing the signal handler that interrupted its call to `sigsuspend()`, its signal mask is the one passed to `sigsuspend()`. It does not get restored to the previous mask until it returns from all the signal handlers that interrupted `sigsuspend()`.
### NAME

sigwait – wait until a signal is posted

### SYNOPSIS

**Default**

```c
#include <signal.h>

int sigwait(sigset_t *set);
```

**POSIX**

```c
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 (see `thr_create(3THR)` or LWP. If no signal in `set` is pending, then `sigwait()` blocks until a signal in `set` becomes pending. The selected signal is cleared from the set of signals pending on the calling thread or LWP and the number of the signal is returned, or in the POSIX version (see `standards(5)`) placed in `sig`. The selection of a signal in `set` is independent of the signal mask of the calling thread or LWP. This means a thread or LWP can synchronously wait for signals that are being blocked by the signal mask of the calling thread or LWP. 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 `sigwait()` is called on an ignored signal, then the occurrence of the signal will be ignored, unless `sigaction()` changes the disposition. If more than one thread or LWP waits for the same signal, only one is unblocked when the signal arrives.

### RETURN VALUES

Upon successful completion, the default version of `sigwait()` returns a signal number; the POSIX version returns 0 and stores the received signal number at the location pointed to by `sig`. Otherwise, −1 is returned and `errno` is set to indicate an error.

### ERRORS

The `sigwait()` function will fail if:

- **EINVAL** The `set` argument contains an unsupported signal number.
- **EFAULT** The `set` argument points to an invalid address.

### 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.

```c
#include <pthread.h>

#include <pthread.h>

CALENDAR(2) 268  man pages section 2: System Calls • Last Revised 24 Jan 1997
```
EXAMPLE 1 Creating a thread to handle receipt of a signal (Continued)

```c
#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;

    thr_setconcurrency(3);
    sigfillset ( &signalSet );
    /*
     * Block signals in initial thread. New threads will
     * inherit this signal mask.
     */
    pthread_sigmask ( SIG_BLOCK, &signalSet, NULL );
    printf("Creating threads\n");
    /* POSIX thread create arguments:
     * thr_id, attr, strt_func, arg
     */
    pthread_create(&t, NULL, sigint, NULL);
    pthread_create(&t2, NULL, threadTwo, NULL);
    pthread_create(&t3, NULL, threadThree, NULL);
    printf("##################
    press CTRL-C to deliver SIGINT to sigint thread\n    ##################
");
    thr_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( ));
    thr_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( ));
    thr_exit((void *)0);
}
static void *
sigint(void *)
{
    return NULL;
}
```

System Calls 269
EXAMPLE 1 Creating a thread to handle receipt of a signal  (Continued)

```c
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( ));
    thr_exit((void *)0);
}

void *
sigint(void *arg)
{
    int    sig;
    int    err;

    printf("thread sigint [tid: %d] awaiting SIGINT\n",
            pthread_self( ));

    /* use POSIX sigwait( ) -- 2 args
     * signal set, signum
     */
    err = sigwait ( &signalSet, &sig );

    /* test for SIGINT; could catch other signals */
    if (err || sig != SIGINT)
        abort( );

    printf("SIGINT signal %d caught by sigint thread [tid: %d]\n",
            sig, pthread_self( ));
    thr_exit((void *)0);
}

SEE ALSO
    sigaction(2), sigpending(2), sigprocmask(2), sigsuspend(2),
    thr_create(THR), thr_sigsetmask(3THR), signal(3HEAD), 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 POSIX
    standard interface.

    In Solaris 2.4 and earlier releases, the call to sigwait() from a multithreaded process
    overrode the signal's ignore disposition; even if a signal's disposition was SIG_IGN, a
    call to sigwait() resulted in catching the signal, if generated. This is unspecified
    behavior from the standpoint of the POSIX 1003.1c specification.
```
In Solaris 2.5, the behavior of `sigwait()` was corrected, so that it does not override the signal’s ignore disposition. This change can cause applications that rely on the old behavior to break. Applications should employ `sigwait()` as follows: Install a dummy signal handler, thereby changing the disposition from `SIG_IGN` to having a handler. Then, any calls to `sigwait()` for this signal would catch it upon generation.
NAME
__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.

<table>
<thead>
<tr>
<th>Trap Name</th>
<th>User Trap Type (utrap_entry_t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>illegal_instruction</td>
<td>UT_ILLTRAP_INSTRUCTION +* or UT_ILLEGAL_INSTRUCTION</td>
</tr>
<tr>
<td>fp_disabled</td>
<td>UT_FP_DISABLED +*</td>
</tr>
<tr>
<td>fp_exception_ieee_754</td>
<td>UT_FP_EXCEPTION_IEEE_754 +</td>
</tr>
<tr>
<td>fp_exception_other</td>
<td>UT_FP_EXCEPTION_OTHER</td>
</tr>
<tr>
<td>tag_overflow</td>
<td>UT_TAG_OVERFLOW +*</td>
</tr>
<tr>
<td>division_by_zero</td>
<td>UT_DIVISION_BY_ZERO +</td>
</tr>
<tr>
<td>mem_address_not_aligned</td>
<td>UT_MEM_ADDRESS_NOT_ALIGNED +</td>
</tr>
</tbody>
</table>
The following explanations are provided for those user trap types that are not self-explanatory.

**UT_IILLTRAP_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. The exception return value will contain either BUS_ADDRERR or BUS_OBJERR.
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.

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD-type (LDSTUB)</td>
<td>%o2 contains the effective address (rs1 + rs2 \ simm13).</td>
</tr>
<tr>
<td>ST-type (CAS, SWAP)</td>
<td>%o2 contains the effective address (rs1 + rs2 \ simm13).</td>
</tr>
<tr>
<td>Integer arithmetic</td>
<td>%o2 contains the rs1 value. %o3 contains the rs2 \ simm13 value. %o4 contains the contents of the %y register.</td>
</tr>
<tr>
<td>Floating-point arithmetic</td>
<td>%o2 contains the address of rs1 value. %o3 contains the address of rs2 value.</td>
</tr>
<tr>
<td>Control-transfer</td>
<td>%o2 contains the target address (rs1 + rs2 \ simm13).</td>
</tr>
</tbody>
</table>
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.

```c
extern void *fpdis_trap_handler();
utrap_handler_t new_precise = (utrap_handler_t)fpdis_trap_handler;
double d;
```
EXAMPLE 1 A sample program using the __sparc_utrap_install() function

(Continued)

int err;
err = __sparc_utrap_install(UT_FP_DISABLED, new_precise,
UTH_NOCHANGE, NULL, NULL);
if (err == EINVAL) {
    /* unexpected error, do something */
    exit (1);
}

d = 1.0e-300;
ENTRY(fpdis_trap_handler)
wr %g0, FPRS_FEF, %fprs
jmp1 %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:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

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.
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.

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:

```c
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*/
```

Descriptions of structure members are as follows:

**st_mode**
The mode of the file as described in mknod(2). In addition to the modes described in mknod(), the mode of a file may also be S_IFLNK if the file is a symbolic link. S_IFLNK may only be returned by lstat().

**st_ino**
This field uniquely identifies the file in a given file system. The pair st_ino and st_dev uniquely identifies regular files.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>st_dev</td>
<td>This field uniquely identifies the file system that contains the file. Its value may be used as input to the <code>ustat()</code> function to determine more information about this file system. No other meaning is associated with this value.</td>
</tr>
<tr>
<td>st_rdev</td>
<td>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.</td>
</tr>
<tr>
<td>st_nlink</td>
<td>This field should be used only by administrative commands.</td>
</tr>
<tr>
<td>st_uid</td>
<td>The user ID of the file’s owner.</td>
</tr>
<tr>
<td>st_gid</td>
<td>The group ID of the file’s group.</td>
</tr>
<tr>
<td>st_size</td>
<td>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 <code>pipe(2)</code>.</td>
</tr>
<tr>
<td>st_atime</td>
<td>Time when file data was last accessed. Changed by the following functions: <code>creat()</code>, <code>mknod()</code>, <code>pipe()</code>, <code>utime(2)</code>, and <code>read(2)</code>.</td>
</tr>
<tr>
<td>st_mtime</td>
<td>Time when data was last modified. Changed by the following functions: <code>creat()</code>, <code>mknod()</code>, <code>pipe()</code>, <code>utime()</code>, and <code>write(2)</code>.</td>
</tr>
<tr>
<td>st_ctime</td>
<td>Time when file status was last changed. Changed by the following functions: <code>chmod()</code>, <code>chown()</code>, <code>creat()</code>, <code>link(2)</code>, <code>mknod()</code>, <code>pipe()</code>, <code>unlink(2)</code>, <code>utime()</code>, and <code>write()</code>.</td>
</tr>
<tr>
<td>st_blksize</td>
<td>A hint as to the &quot;best&quot; unit size for I/O operations. This field is not defined for block special or character special files.</td>
</tr>
<tr>
<td>st_blocks</td>
<td>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.</td>
</tr>
</tbody>
</table>

**RETURN VALUES**

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

**ERRORS**

The `stat()`, `fstat()`, and `lstat()` functions will fail if:

- `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()` and `lstat()` 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
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 the 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.

EOVERFLOW
A component is too large to store in the structure pointed to by buf.

The fstat() function will fail if:
EBADF
The fildes argument is not a valid open file descriptor.
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.
EOVERFLOW
A component is too large to store in the structure pointed to by buf.

USAGE
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:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>stat() and fstat() are Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
chmod(2), chown(2), creat(2), link(2), mknod(2), pipe(2), read(2), time(2), unlink(2), utime(2), write(2), fattach(3C), attributes(5), lf64(5), stat(3HEAD)

NOTES
If you use chmod(2) to change the file group owner permissions on a file with ACL entries, both the file group owner permissions and the ACL mask are changed to the new permissions. Be aware that the new ACL mask permissions may change the effective permissions for additional users and groups who have ACL entries on the file.
statvfs(2)

NAME      statvfs, fstatvfs – get file system information

SYNOPSIS  #include <sys/types.h>
#include <sys/statvfs.h>

    int statvfs(const char *path, struct statvfs *buf);
    int fstatvfs(int fildes, struct statvfs *buf);

DESCRIPTION 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-super-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-super-user*/
    u_long f_fsid; /* file system id (dev for now) */
    char f_basetype[FSTYPSZ]; /* target fs type name,
                              null-terminated */
    u_long f_flag; /* bit mask of flags */
    u_long f_namemax; /* maximum file name length */
    char f_fstr[32]; /* file system specific string */
    u_long f_filler[16]; /* reserved for future expansion */

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.
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` 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 `1f64(5)`.

**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), 1f64(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) time value.
         | EPERM      The effective user of the calling process is not super-user.

SEE ALSO | time(2)
### NAME
swapctl – manage swap space

### SYNOPSIS
```c
#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.
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.

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 effective user of the calling process is not super-user.
- **EROFS**: The pathname specified for SC_ADD is a read-only file system.

### Return Values

**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 effective user of the calling process is not super-user.

**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.

**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.

```c
#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) {
      perror("swapctl");
      exit(1);
    }
    again:
}
```

**EXAMPLES**

System Calls 285
EXAMPLE 1 The usage of the SC_GETNSWP and SC_LIST commands.  

(Continued)

```c
} 
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);
```
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    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.

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.

SEE ALSO cp(1), link(2), open(2), readlink(2), stat(2), unlink(2)
### NAME
sync – update super block

### SYNOPSIS
```c
#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.

### SEE ALSO
`df(1M)`, `fsck(1M)`, `fsync(3C)`
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 `fname`, 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.

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.

The `sysfs()` function will fail if:

- **EFAULT**: The `buf` or `fname` argument points to an illegal address.
- **EINVAL**: The `fname` 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
```c
#include <sys/systeminfo.h>

long 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:

<table>
<thead>
<tr>
<th><code>command</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SI_SYSNAME</td>
<td>Copy into the array pointed to by <code>buf</code> the string that would be returned by <code>uname(2)</code> in the <code>sysname</code> field. This is the name of the implementation of the operating system, for example, SunOS or UTS.</td>
</tr>
<tr>
<td>SI_HOSTNAME</td>
<td>Copy into the array pointed to by <code>buf</code> a string that names the present host machine. This is the string that would be returned by <code>uname(2)</code> in the <code>nodename</code> field. This hostname or nodename is often the name the machine is known by locally. The <code>hostname</code> is the name of this machine as a node in some network. Different networks may have different names for the node, but presenting the nodename to the appropriate network directory or name-to-address mapping service should produce a transport end point address. The name may not be fully qualified. Internet host names may be up to 256 bytes in length (plus the terminating null).</td>
</tr>
<tr>
<td>SI_SET_HOSTNAME</td>
<td>Copy the null-terminated contents of the array pointed to by <code>buf</code> into the string maintained by the kernel whose value will be returned by succeeding calls to <code>sysinfo()</code> with the command <code>SI_HOSTNAME</code>. This command requires that the effective-user-id be super-user.</td>
</tr>
<tr>
<td>SI_RELEASE</td>
<td>Copy into the array pointed to by <code>buf</code> the string that would be returned by <code>uname(2)</code> in the <code>release</code> field. Typical values might be 5.2 or 4.1.</td>
</tr>
<tr>
<td>SI_VERSION</td>
<td>Copy into the array pointed to by <code>buf</code> the string that would be returned by <code>uname(2)</code> in the <code>version</code> field. The syntax and semantics of this string are defined by the system provider.</td>
</tr>
<tr>
<td><strong>SI_MACHINE</strong></td>
<td>Copy into the array pointed to by <code>buf</code> the string that would be returned by <code>uname(2)</code> in the <code>machine</code> field, for example, <code>sun4c</code>, <code>sun4d</code>, or <code>sun4m</code>.</td>
</tr>
<tr>
<td><strong>SI_ARCHITECTURE</strong></td>
<td>Copy into the array pointed to by <code>buf</code> a string describing the basic instruction set architecture of the current system, for example, <code>sparc</code>, <code>mc68030</code>, <code>m32100</code>, or <code>i386</code>. These names may not match predefined names in the C language compilation system.</td>
</tr>
<tr>
<td><strong>SI_ISALIST</strong></td>
<td>Copy into the array pointed to by <code>buf</code> 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 may 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 <code>isalist(5)</code>; these names may or may not match predefined names or compiler options in the C language compilation system.</td>
</tr>
<tr>
<td><strong>SI_PLATFORM</strong></td>
<td>Copy into the array pointed to by <code>buf</code> a string describing the specific model of the hardware platform, for example, <code>SUNW,Sun_4_75</code>, <code>SUNW,SPARCSистем-600</code>, or <code>i86pc</code>.</td>
</tr>
<tr>
<td><strong>SI_HW_PROVIDER</strong></td>
<td>Copies the name of the hardware manufacturer into the array pointed to by <code>buf</code>.</td>
</tr>
<tr>
<td><strong>SI_HW_SERIAL</strong></td>
<td>Copy into the array pointed to by <code>buf</code> a string which is the ASCII representation of the hardware-specific serial number of the physical machine on which the function is executed. Note that this may be implemented in Read-Only Memory, using software constants set when building the operating system, or by other means, and may contain non-numeric characters. It is anticipated that manufacturers will not issue the same “serial number” to more than one physical machine. The pair of strings returned by <code>SI_HW_PROVIDER</code> and...</td>
</tr>
</tbody>
</table>
SI_HW_SERIAL is likely to be unique across all vendor’s SVR4 implementations.

SI_SRPC_DOMAIN
Copies the Secure Remote Procedure Call domain name into the array pointed to by buf.

SI_SET_SRPC_DOMAIN
Set the string to be returned by sysinfo() with the SI_SRPC_DOMAIN command to the value contained in the array pointed to by buf. This command requires that the effective-user-id be super-user.

SI_DHCP_CACHE
Copy into the array pointed to by buf an ASCII string consisting of the ASCII hexadecimal 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.

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.

The sysinfo() function will fail if:

EFAULT
The buf argument does not point to a valid address.

EINVAL
The data for a SET command exceeds the limits established by the implementation.

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

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), uname(2), gethostid(3C), gethostname(3C), sysconf(3C), isalist(5), standards(5)
time(2)

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:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  
stime(2), ctime(3C), attributes(5)
times – get process and child process times

#include <sys/times.h>
#include <limits.h>

clock_t times(struct tms *buffer);

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(2) or waitpid(2) 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.

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.

The times() function will fail if:

EFAULT The buffer argument points to an illegal address.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>
times(2)

SEE ALSO  time(1), timex(1), exec(2), fork(2), time(2), wait(2), waitid(2), waitpid(2), attributes(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.

As specified by cmd, the following commands are available:

A_SHUTDOWN The system is shut down. All user processes are killed, the buffer cache is flushed, and the root file system is unmounted. The action to be taken after the system has been shut down is specified by fcn. The functions are generic; the hardware capabilities vary on specific machines.
   AD_HALT Halt the processor(s).
   AD_POWEROFF Halt the processor(s) and turn off the power.
   AD_BOOT Reboot the system, using the kernel file.
   AD_IBOOT Interactive reboot; user is prompted for bootable program name.

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

A_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 three subcommands are available.
   AD_COMPRESS Save the system state to the state file with compression of data.
   AD_CHECK Check if your system supports suspend and resume. Without performing a system suspend/resume, this command checks if this feature is currently available on your system.
   AD_FORCE Force AD_COMPRESS even when threads of drivers are not suspendable.

RETURN VALUES
Upon successful completion, the value returned depends on cmd as follows:
A_SHUTDOWN Never returns.
A_REBOOT Never returns.
A_FREEZE 0 upon resume.
A_REMOUNT 0.

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

ERRORS The uadmin() function will fail if:
EPERM The effective user of the calling process is not super-user.
ENOMEM Suspend/resume ran out of physical memory.
ENOSPC Suspend/resume could not allocate enough space on the root file
system to store system information.
ENOTSUP Suspend/resume not supported on this platform.
ENXIO Unable to successfully suspend system.
EBUSY Suspend already in progress.

SEE ALSO dumpadm(1M), kernel(1M), uadmin(1M)
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_GMMLIM**: 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 not having appropriate privileges attempts 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).

SEE ALSO | brk(2), getrlimit(2), write(2)
umask(2)

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:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO mkdir(1), sh(1), intro(2), chmod(2), creat(2), mknod(2), open(2), stat(3HEAD), attributes(5)
umount(2)

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 may resort 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. Currently only nfs- and ufs-type file systems 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:

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.
ENOENT The file pointed to by file does not exist.
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.
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 process’s effective user ID is not superuser.
The file pointed to by file is remote.

The umount2() function will fail if:

EREMOTE The file pointed to by file is remote.

ENOTSUP The file pointed to by file does not support this operation.

USAGE The umount() and umount2() functions may be invoked only by the superuser. Because it provides greater functionality, the umount2() function is preferred.

SEE ALSO mount(2)
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:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
`uname(1), sysinfo(2), sysconf(3C), attributes(5)"
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.

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 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 `unlink()` returns, but the removal of the file contents will be postponed until all references to the file are closed.

The `path` argument must not name a directory unless the process has appropriate privileges and the implementation supports using `unlink()` on directories.

Upon successful completion, `unlink()` 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.

Upon successful completion, 0 is returned. Otherwise, −1 is returned, `errno` is set to indicate the error, and the file is not unlinked.

The `unlink()` function will fail if:

- **EACCES** Search permission is denied for a component of the `path` prefix; write permission is denied on the directory containing the link to be removed; the parent directory has the sticky bit set and the file is not writable by the user; or the user does not own the parent directory and the user does not own the file.

- **EBUSY** The entry to be unlinked is the mount point for a mounted file system.

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

- **EINVAL** 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` component exceeds NAME_MAX while _POSIX_NO_TRUNC is in effect.

- **ENOENT** The named file does not exist or is a null pathname.
The unlink() function 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:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO** rm(1), close(2), link(2), open(2), rmdir(2), remove(3C), attributes(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)

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 the super-user may 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; or the effective user ID of the process is not super-user and not the owner of the file, write permission is denied for the file, and times is NULL.

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

A component of the path prefix is not a directory.

The effective user of the calling process is not super-user and not the owner of the file, and times is not NULL.

The file system containing the file is mounted read-only.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

See also stat(2), attributes(5)
NAME | utimes – set file access and modification times

SYNOPSIS | #include <sys/time.h>

        int utimes(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 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 super-user privileges 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() function 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.
| 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.
utimes(2)

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.

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() function may fail if:

ENAMETOOLONG
Path name resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.

SEE ALSO
stat(2)
vfork(2)

NAME
vfork – spawn new process in a virtual memory efficient way

SYNOPSIS
#include <unistd.h>

pid_t vfork(void);

DESCRIPTION
The vfork() function creates new processes without fully copying the address space
of the old process. This function is useful in instances where the purpose of a fork(2)
operation would be 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)). The parent process is suspended while the child is using its
resources.

In a multithreaded application, vfork() borrows only the thread of control that
called vfork() in the parent; that is, the child contains only one thread. In that sense,
vfork() behaves like fork().

The vfork() function can normally be used the same way as fork(). The procedure
that called vfork(), however, should not return while running in the child’s context,
since the eventual return from vfork() 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 flush and close standard I/O
channels, and thereby corrupt the parent process’s standard I/O data structures. The
_exit() function should be used even with fork() to avoid flushing the buffered
data twice.

RETURN VALUES
Upon successful completion, vfork() returns 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() function 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.

SEE ALSO
exec(2), exit(2), fork(2), ioctl(2), wait(2), exit(3C)

NOTES
The use of vfork() for any purpose other than as a prelude to an immediate call to a
function from the exec family or to _exit() is not advised.

The vfork() function is unsafe in multithreaded applications.

This function will be eliminated in a future release. The memory sharing semantics of
vfork() can be obtained through other mechanisms.

System Calls 311
To avoid a possible deadlock situation, processes that are children in the middle of a \texttt{vfork()} are never sent \texttt{SIGTTOU} or \texttt{SIGTTIN} signals; rather, output or ioctl's are allowed and input attempts result in an EOF indication.

On some systems, the implementation of \texttt{vfork()} causes the parent to inherit register values from the child. This can create problems for certain optimizing compilers if \texttt{<unistd.h>} is not included in the source calling \texttt{vfork()}. 
NAME  vhangup – virtually “hangup” the current controlling terminal

SYNOPSIS  
```c
#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.
wait(2)

NAME wait – wait for child process to stop or terminate

SYNOPSIS
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *stat_loc);

DESCRIPTION
The wait() function will suspend execution of the calling thread until status
information for one of its terminated child processes is available, or until delivery of a
signal whose action is either to execute a signal-catch function or to terminate the
process. If more than one thread is suspended in wait() or waitpid(2) awaiting
termination of the same process, exactly one thread will return the process status at
the time of the target process termination. If status information is available prior to the
call to wait(), return will be immediate.

If wait() returns because the status of a child process is available, it returns the
process ID of the child process. If the calling process specified a non-zero value for
stat_loc, the status of the child process is stored in the location pointed to by stat_loc.
That status may be evaluated with the macros described on the wstat(3XFN) manual
page.

In the following, status is the object pointed to by stat_loc:

- If the child process stopped, the high order 8 bits of status will contain the number
  of the signal that caused the process to stop and the low order 8 bits will be set
  equal to WSTOPFLG.
- If the child process terminated due to an _exit() call, the low order 8 bits of
  status will be 0 and the high order 8 bits will contain the low order 8 bits of the
  argument that the child process passed to _exit(); see exit(2).
- If the child process terminated due to a signal, the high order 8 bits of status will be
  0 and the low order 8 bits will contain the number of the signal that caused the
  termination. In addition, if WCOREFLG is set, a “core image” will have been
  produced; see signal(3HEAD) and wstat(3XFN).

If the calling process has SA_NOCLDWAIT set or has SIGCHLD set to SIG_IGN, and the
process has no unwaited children that were transformed into zombie processes, it will
block until all of its children terminate, and wait() will fail and set errno to
ECHILD.

If a parent process terminates without waiting for its child processes to terminate, the
parent process ID of each child process is set to 1, with the initialization process
inheriting the child processes; see intro(3).

RETURN VALUES
When wait() returns due to a terminated child process, the process ID of the child is
returned to the calling process. Otherwise, −1 is returned and errno is set to indicate
the error.

ERRORS The wait() function will fail if:

ECHILD The calling process has no existing unwaited-for child processes.

314 man pages section 2: System Calls • Last Revised 11 Feb 1999
The function was interrupted by a signal.

Since `wait()` blocks on a stopped child, a calling process wishing to see the return results of such a call should use `waitid(2)` or `waitpid(2)` instead of `wait()`. See `attributes(5)` for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO `intro(3), exec(2), exit(2), fork(2), pause(2), ptrace(2), waitid(2), waitpid(2), signal(3C), attributes(5), signal(3HEAD), wstat(3XFN)`.  

System Calls 315
waitid(2)

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(2)).

The infop argument must point to a siginfo_t structure, as defined in siginfo3(HEAD). 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.

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 idtype equal to P_ALL and options equal to WEXITED | WTRAPPED, waitid() is equivalent to wait(2).

**SEE ALSO** intro(3), exec(2), exit(2), fork(2), pause(2), ptrace(2), sigaction(2), wait(2), signal(3C), siginfo(3HEAD)
waitpid – wait for child process to change state

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

pid_t waitpid(pid_t pid, int *stat_loc, int options);

The waitpid() function will suspend execution of the calling thread until status information for one of its terminated child processes is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. If more than one thread is suspended in waitpid() or wait(2) awaiting termination of the same process, exactly one thread will return the process status at the time of the target process termination. If status information is available prior to the call to waitpid(), return will be immediate.

The pid argument specifies a set of child processes for which status is requested, as follows:

- If pid is equal to (pid_t)-1, status is requested for any child process.
  - If pid is greater than (pid_t)0, it specifies the process ID of the child process for which status is requested.
- If pid is equal to (pid_t)0 status is requested for any child process whose process group ID is equal to that of the calling process.
- If pid is less than (pid_t)-1, status is requested for any child process whose process group ID is equal to the absolute value of pid.

If the calling process has SA_NOCLDWAIT set or has SIGCHLD set to SIG_IGN and the process has no unwaited children that were transformed into zombie processes, it will block until all of its children terminate, and waitpid() will fail and set errno to ECHILD.

If waitpid() returns because the status of a child process is available, then that status may be evaluated with the macros defined by wstat(3XFN). If the calling process had specified a non-zero value of stat_loc, the status of the child process will be stored in the location pointed to by stat_loc.

The options argument is constructed from the bitwise inclusive OR of zero or more of the following flags, defined in the header <sys/wait.h>:

- WCONTINUED The status of any continued child process specified by pid, whose status has not been reported since it continued, is also reported to the calling process.
- WNOHANG waitpid() will not suspend execution of the calling process if status is not immediately available for one of the child processes specified by pid.
- WNOWAIT Keep the process whose status is returned in stat_loc in a waitable state. The process may be waited for again with identical results.
The status of any child processes specified by `pid` that are stopped, and whose status has not yet been reported since they stopped, is also reported to the calling process.

If `waitpid()` returns because the status of a child process is available, it returns a value equal to the process ID of the child process for which status is reported. If `waitpid()` returns due to the delivery of a signal to the calling process, −1 is returned and `errno` is set to EINTR. If `waitpid()` was invoked with `WNOHANG` set in `options`, it has at least one child process specified by `pid` for which status is not available, and status is not available for any process specified by `pid`, then 0 is returned. Otherwise, −1 is returned and `errno` is set to indicate the error.

The `waitpid()` function will fail if:

**ECHILD** The process or process group specified by `pid` does not exist or is not a child of the calling process or can never be in the states specified by `options`.

**EINTR** The `waitpid()` function was interrupted due to the receipt of a signal sent by the calling process.

**EINVAL** An invalid value was specified for `options`.

With `options` equal to 0 and `pid` equal to `(pid_t)−1`, `waitpid()` is identical to `wait(2)`.

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

```
<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>
```

**SEE ALSO** `intro(3), exec(2), exit(2), fork(2), pause(2), ptrace(2), sigaction(2), wait(2), signal(3C), attributes(5), siginfo(3HEAD), wstat(3XFN)`
write(2)

NAME  write, pwrite, writev – write on a file

SYNOPSIS

```c
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t nbyte);

ssize_t pwrite(int fd, const void *buf, size_t nbyte, off_t offset);

#include <sys/uio.h>
ssize_t writev(int fd, 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, `fd`. 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 `fd`. 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` flag of the file status flags is set and `fd` refers to a regular file, a successful `write()` does not return until the data is delivered to the underlying hardware.

If `fd` 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 `fd`.

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 the value of nbyte is greater than SSIZE_MAX, the result is implementation-
dependent.

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(3) 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 performs the same action as write(), except that it writes into a given position without changing the file pointer. The first three arguments to pwrite() are the same as write() with the addition of a fourth argument offset for the desired position inside the file.

**writev()**

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(3) for a definition of {IOV_MAX}. 
The `iovec` structure contains the following members:

```c
    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 `fd` 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 `fd`. 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 `nbyte` of free space is available.

- **EBADF**  
  The `fd` 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.

- **EFAULT**  
  The `buf` argument points to an illegal address.

- **EFBIG**  
  An attempt is made to write a file that exceeds the process’s file size limit or the maximum file size (see `getrlimit(2)` and `ulimit(2)`).

- **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 `fd`.

- **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 STOP 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 \texttt{fdes} 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 \texttt{socket}(3SOCKET), using type \texttt{SOCK\_STREAM} that is no longer connected to a peer endpoint). A SIGPIPE signal will also be sent to the process. 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 \texttt{fdes}.

The \texttt{pwrite()} function fails and the file pointer remains unchanged if:

### ESPIPE
The \texttt{fdes} argument is associated with a pipe or FIFO.

The \texttt{writev()} function will fail if:

### EINVAL
The sum of the \texttt{iov\_len} values in the \texttt{iov} array would overflow an \texttt{ssize\_t}.

The \texttt{write()} and \texttt{writev()} functions may fail if:

### EINVAL
The STREAM or multiplexer referenced by \texttt{fdes} 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, \texttt{errno} is set to the value included in the error message.

The \texttt{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 `int`.

USAGE    The `pwrite()` function has a transitional interface for 64-bit file offsets. See `lfs64(5)`.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td><code>write()</code> is Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO    `Intro(3), 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), lfs64(5), streamio(7))`
yield(2)

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(3THR)
Index

Numbers and Symbols
install a SPARC V9 user trap handler, 272

A
access — determine accessibility of a file, 40
access permission mode of file, change —
chmod, 59
accounting, enable or disable process accounting — acct, 42
acct — enable or disable process accounting, 42
acl — get or set a file’s Access Control List (ACL), 43
adjtime — correct the time to allow synchronization of the system clock, 45
adjust local clock parameters —
ntp_adjtime, 175
alarm — set a process alarm clock, 46
audit — write an audit record, 48
auditon — manipulate auditing, 49
auditsvc() function, 53

B
bind LWPs to a processor —
processor_bind, 202
bind LWPs to a set of processors —
pset_bind, 207
brk — change the amount of space allocated for the calling process’s data segment, 55

C
chdir — change working directory, 57
child processes
allows a parent process to control the execution of a child process — ptrace, 212
get time — times, 295
wait for child process to change state —
waitid, 316
wait for child process to change state —
waitpid, 318
wait for child process to stop or terminate —
wait, 314
chmod — change access permission mode of file, 59
chown — change owner and group of a file, 62
chroot — change root directory, 65
clock, get local clock values — ntp_gettime, 176
continue or suspend LWP execution
— _lwp_continue, 139
— _lwp_suspend, 139
CPU-use, process execution time profile —
profil, 205
creat — create a new file or rewrite an existing one, 69
create a new process — fork, 89
fork1, 89
create a new light-weight process —
_lwp_create, 129
create session and set process group ID —
setsid, 243
**D**

deliver process signals to specific LWPs
  — _lwp_sigsigredirect, 260
  — _signotifywait, 260
determine accessibility of a file — access, 40
determine accessibility of a file — access, 40
directories
  change working directory — chdir, 57
directories
  create a new one — mknod, 148
directories
  get configurable pathname variables —
    pathconf, 92
directories
  make a new one — mkdir, 146
directories
  read directory entries and put in a file system
    independent format — getdents, 101
directories
  remove — rmdir, 227
directories
  discover all message queue identifiers —
    msgids, 164
directories
  discover all shared memory identifiers —
    shmids, 251
directories
  dup — duplicate an open file descriptor, 71

**E**
effective group ID, set — setregid(), 241
effective user ID, set — setreuid(), 242
eexec — execute a file, 72
eexec — execute a file, 72
eexec — execute a file, 72
eexec — execute a file, 72
eexec — execute a file, 72
eexec — execute a file, 72
execve — execute a file, 78
_exit — terminate process, 78
_exit — terminate process, 78

**F**
fac — get or set a file’s Access Control List
  (ACL), 43
fchdir — change working directory, 57
fchmod — change access permission mode of
    file, 59
fchown — change owner and group of a file, 62
fcntl — file control, 81
file control — fcntl, 81
file descriptor, duplicate an open one —
  dup, 71
file pointer, read/write
  move — lseek, 122, 123
file status, get — stat, lstat, fstat, 277
file system
  get information — statvfs, fstatvfs, 280
  get statistics — usstat, 306
  make a symbolic link to a file —
    symlink, 287
  remove link — unlink, 304
  returns information about the file system
    types configured in the system —
    sysfs, 290
  update super block — sync, 289
files
  change access permission mode of file —
    chmod, 59
  change owner and group of a file —
    chown, 62
  change the name of a file — rename, 223
  create a new file or rewrite an existing one —
    creat, 69
  execute — exec, 72
  get configurable pathname variables —
    pathconf, 92
  link to a file — link, 120
  move read/write file pointer — lseek, 122
    123
  set file access and modification times —
    utime, 307
fork — create a new process, 89
fork — create a new process, 89
fork — spawn new process in a virtual memory
  efficient way — vfork, 311
fork1 — create a new process, 89
fpathconf — get configurable pathname
  variables, 92
fstat — get status on open file known by file
  descriptor, 277
fstatvfs — get file system information, 280

**G**
get and set process audit information —
  getaudit, 97
get and set process audit information —
  getaudit_addr, 97
get and set process audit information —
setaudit, 97
get and set process audit information —
setaudit_addr, 97
get, put, or write extended accounting data —
getacct, 95
get, put, or write extended accounting data —
putacct, 95
get, put, or write extended accounting data —
wracct, 95
get and set process limits — ulimit, 299
get information about a processor set —
set_info, 211
get LWP identifier — _lwp_self, 136
get or change processor operational status —
p_online, 189
get or set a file’s Access Control List (ACL) —
    acl, 43
    facl, 43
get process group ID of session leader —
getsid, 114
getacct — get, put, or write extended
    accounting data, 95
getaudit — get and set process audit
    information, 97
getaudit_addr — get and set process audit
    information, 97
getauid — get user audit identity, 99
getdents — read directory entries and put in a
    file system independent format, 101
getegid — get effective group ID, 115
geteuid — get effective user ID, 115
getgid — get real group ID, 115
getgroups — get supplementary group access
    list IDs, 102
getitimer — get value of interval timer, 103
getmsg — get next message off a stream, 107
getpgid — get process group IDs, 110
getpgrp — get process group IDs, 110
getpmsg — get next message off a stream, 107
getppid — get parent process IDs, 110
getprojid — set or get task or project IDs, 244
getrlimit — control maximum system resource
    consumption, 111
getsid — get process group ID of session
    leader, 114
gettaskid — set or get task or project IDs, 244
getuid — get real user ID, 115
group ID, set real and effective —
    setregid(), 241
group IDs —
    get — getgid, getegid, 115
    set — setgid, 245
    supplementary group access list IDs —
        getgroups, setgroups, 102

H
halt system, — uadmin, 297
hangup signal, the current controlling terminal
    — vhangup, 313

I
I/O
    audit — audit, 48
    multiplexing — poll, 186
initialize an LWP context —
    _lwp_makecontext, 134
interprocess communication, — pipe, 185
interval timer, get or set value of interval timer
    — getitimer, setitimer, 103
ioctl — control device, 116

K
kill — send a signal to a process or a group of
    processes, 118

L
lchown — change owner and group of a file, 62
link — link to a file, 120
link, remove — unlink, 304
link, symbolic, make one to a file —
    symlink, 287
lseek — move extended read/write file
    pointer, 122
lseek — move read/write file pointer, 123
lstat — get status on symbolic link file, 277
LWP, scheduler control — priocntl, 191
man pages section 2: System Calls • February 2004

M
make a directory, or a special or ordinary file — mknod, 148
manage sets of processors
   — pset_assign, 209
   — pset_create, 209
   — pset_destroy, 209

manipulate auditing — auditon, 49
masks, set and get file creation mask — umask, 300
memcntl — memory management control, 141
memory, management control — memcntl, 141
memory, shared
   control operations — shmtl, 247
   get segment identifier — sjmget, 249
   operations — shmop, 253
memory management, change the amount of space allocated for the calling process’s data segment — brk, sbrk, 55
memory mapping, set protection — mprotect, 160
memory pages
   determine residency — mincore, 145
   map — mmap, 151
   unmap — munmap, 173
message control operations, — msgctl, 161
message queue, get — msgget, 163
message queue snapshot operation — msgsnap, 168
message receive operation — msgrcv, 166
message send operation — msgsnd, 171
messages, send a message on a stream — putmsg, 214
mincore — determine residency of memory pages, 145
mkdir — make a directory, 146
mknod — make a directory, or a special or ordinary file, 148
mmap — map pages of memory, 151
mount — mount a file system, 157
mount a file system — mount, 157
mprotect — set protection of memory mapping, 160
msgctl — message control operations, 161
msgget — get message queue, 163
msgids — discover all message queue identifiers, 164
msgrecv — message receive operation, 166
msgsnd — message send operation snapshot operation, 168
munmap — unmap pages of memory, 171
munmap — unmap pages of memory, 173
mutual exclusion
   — _lwp_mutex_lock, 135
   — _lwp_mutex_trylock, 135

_lwp_cond_broadcast — signal a condition variable, 125
_lwp_cond_reltimedwait — wait on a condition variable, 126
_lwp_cond_signal — signal a condition variable, 125
_lwp_cond_timedwait — wait on a condition variable, 126
_lwp_cond_wait — wait on a condition variable, 126
_lwp_continue — continue or suspend LWP execution, 139
_lwp_create — create a new light-weight process, 129
_lwp_exit — terminate the calling LWP, 131
_lwp_getprivate — set/get LWP specific storage, 138
_lwp_info — return the time-accounting information of a single LWP, 132
_lwp_kill — send a signal to a LWP, 133
_lwp_makecontext — initialize an LWP context, 134
_lwp_mutex_lock — mutual exclusion, 135
_lwp_mutex_trylock — mutual exclusion, 135
_lwp_mutex_unlock — mutual exclusion, 135
_lwp_self — get LWP identifier, 136
_lwp_sema_init — semaphore operations, 137
_lwp_sema_post — semaphore operations, 137
_lwp_sema_trywait — semaphore operations, 137
_lwp_sema_wait — semaphore operations, 137
_lwp_setprivate — set/get LWP specific storage, 138
_lwp_sigredirect — deliver process signals to specific LWP’s, 260
_lwp_suspend — continue or suspend LWP execution, 139
_lwp_wait — wait for a LWP to terminate, 140

man pages section 2: System Calls • February 2004
mutual exclusion (Continued)
— _lwp_mutex_unlock, 135

N
nice — change priority of a time-sharing
process, 174
ntp_adjtime — adjust local clock
parameters, 175
ntp_gettime — get local clock values, 176

O
open — open a file, 177
open a file — open, 177
operating system, get name of current one —
uname, 303
owner of file, change — chown, 62

P
p_online — get or change processor operational
status, 189
pathconf — get configurable pathname
variables, 92
pathname, get configurable variables —
pathconf, 92
pause — suspend process until signal, 183
pipe — create an interprocess channel, 185
poll — input/output multiplexing, 186
pread — read from file, 217
priocntl — process scheduler control, 191
priocntlset — generalized process scheduler
control, 200
process, time-sharing, change priority —
nice, 174
process accounting, enable or disable —
acct, 42
process alarm clock, set — alarm, 46
process group ID
set — setpgid, 239
set — setpgrp, 240
process scheduler
control — priocntl, 191
generalized control — priocntlset, 200
process statistics, process execution time profile
— profil, 205
processes
allows a parent process to control the
execution of a child process — ptrace, 212
change priority of a time-sharing process —
nice, 174
create a new one — fork, 89
create an interprocess channel — pipe, 185
execute a file — exec, 72
execution time profile — profil, 205
generalized scheduler control —
priocntlset, 200
get identification — getpwnam, getpgrp, geteuid,
getegid, 110
get next message off a stream — getmsg, 107
get or set value of interval timer — getitimer,
setitimer, 103
g et real user, effective user, real group, and
effective group IDs — getuid, geteuid,
getgid, getegid, 115
get time — times, 295
read from file — read, 217
read directory entries and put in a file system
independent format — getdents, 101
send a signal to a process or a group of
processes — kill, 118
set a process alarm clock — alarm, 46
set and get file creation mask — umask, 300
set process group ID — setpgid, 239
set process group ID — setpgrp, 240
spawn new process in a virtual memory
efficient way — vfork, 311
supplementary group access list IDs —
getgroups, setgroups, 102
suspend process until signal — pause, 183
the current controlling terminal —
vhangup, 313
wait for child process to change state —
waitid, 316
wait for child process to change state —
waitpid, 318
wait for child process to stop or terminate —
wait, 314
processes and protection
— setregid(), 241
— setreuid(), 242
processor_bind — bind LWPs to a processor, 202
processor_info — determine type and status of a processor, 204
prof — process execution time profile, 205
profiling utilities, execution time profile — prof, 205
pset_assign — manage sets of processors, 209
pset_bind — bind LWPs to a set of processors, 207
pset_create — manage sets of processors, 209
pset_destroy — manage sets of processors, 209
pset_info — get information about a processor set, 211
ptrace — allows a parent process to control the execution of a child process, 212
putacct — get, put, or write extended accounting data, 95
putmsg — send a message on a stream, 214
putpmsg — send a message on a stream, 214
pwrite — write on a file, 320

R
read from file — read, 217
pread, 217
readv, 217
read the contents of a symbolic link — readlink, 222
read/write file pointer
move — lseek, 122, 123
readlink — read the contents of a symbolic link, 222
read — read from file, 217
real group ID, set — setregid(), 241
real user ID, set — setreuid(), 242
reboot system, — uadmin, 297
remount root file system, — uadmin, 297
rename — change the name of a file, 223
resolve all symbolic links of a path name — resolve, 226
resolvepath — resolve all symbolic links of a path name, 226
rmdir — remove a directory, 227
root directory, change — chroot, 65

S
sbrk — change the amount of space allocated for the calling process’s data segment, 55
semaphore operations — semop, 236
semaphore operations — semtimedop, 236
semaphore operations
— _lwp_sema_init, 137
— _lwp_sema_post, 137
— _lwp_sema_trywait, 137
— _lwp_sema_wait, 137
semaphores
control operations — semctl, 229
get a set — semget, 232
semctl — semaphore control operations, 229
semget — get set of semaphores, 232
semop — semaphore operations, 236
semtimedop — semaphore operations, 236
send a signal to a LWP — _lwp_kill, 133
set or get task or project IDs — getprojid, 244
set or get task or project IDs — gettaskid, 244
set or get task or project IDs — settaskid, 244
set file access and modification times — utimes, 309
set/get LWP specific storage
— _lwp_getprivate, 138
— _lwp_setprivate, 138
setaudit — get and set process audit information, 97
setaudit_addr — get and set process audit information, 97
setauditid — set user audit identity, 99
setegid — set effective group ID, 245
seteuid — set effective user ID, 245
setgid — set group ID, 245
setgroups — set supplementary group access list IDs, 102
setitimer — set value of interval timer, 103
setpgid — set process group ID, 239
setpgrp — set process group ID, 240
setregid() — set real and effective group ID, 241
setreuid() — set real and effective user IDs, 242
setrlimit — control maximum system resource consumption, 111
setsid — create session and set process group ID, 243
settaskid — set or get task or project IDs, 244
setuid — set user ID, 245
shared memory
  control operations — shmctl, 247
  get segment identifier — sjmget, 249
operations — shmp, 253
shmctl — shared memory control
operations, 247
shmget — get shared memory segment
identifier, 249
shmids — discover all shared memory
identifiers, 251
shmop — shared memory operations, 253
shutdown, — uadmin, 297
sigaction — detailed signal management, 255
sigaltstack — set or get signal alternate stack
context, 258
signal a condition variable
  — _lwp_cond_broadcast, 125
  — _lwp_cond_signal, 125
signal alternate stack, set or get context —
sigaltstack, 258
signal management, detailed — sigaction, 255
signal mask
  change and/or examine — sigprocmask, 263
  install, and suspend process until signal —
sigsuspend, 266
signals, examine blocked and pending ones —
sigpending, 262
  _signotifywait — deliver process signals to
  specific LWPs, 260
sigpending — examine signals that are blocked
and pending, 262
sigprocmask — change and/or examine calling
  process’s signal mask, 263
sigsend — send a signal to a process or a group
  of processes, 264
sigsendset — provides an alternate interface to
  sigsend for sending signals to sets of
  processes, 264
sigsuspend — install a signal mask and
  suspend process until signal, 266
sigwait() — wait until a signal is posted, 268
special files, create a new one — mknod, 148
stat — get file status, 277
statistics, get for mounted file system —
  ustat, 306
statvfs — get file system information, 280
stime — set system time and date, 282

STREAMS
  get next message off a stream — getmsg, 107
I/O control functions — ioctl, 116
  send a message on a stream — putmsg, 214
super block, update — sync, 289
swap space, manage — swapctl, 283
swapctl — manage swap space, 283
symbolic link, make one to a file —
symlink, 287
symlink — make a symbolic link to a file, 287
sync — update super block, 289
sysinfo — get and set system information
  strings, 291
system administration, administrative control
  — uadmin, 297
system clock, synchronization — adjtime, 45
system information, get and set strings —
sysinfo, 291
system operation, update super block —
sync, 289
system resources, control maximum system
  resource consumption — getrlimit,
  setrlimit, 111

T
terminate process
  — _exit, 78
  — exit, 78
terminate the calling LWP — _lwp_exit, 131
time — get time, 294
time
  correct the time to allow synchronization of
  the system clock — adjtime, 45
  set system time and date — stime, 282
time-accounting, single LWP — _lwp_info, 132
times — get process and child process
  times, 295

U
ulimit — get and set process limits, 299
umask — set and get file creation mask, 300
umount — unmount a file system, 301
umount2 — unmount a file system, 301
uname — get name of current operating system, 303
unlink — remove directory entry, 304
unmount a file system — umount2, 301
unmount a file system — umount, 301
user audit identity
  get user audit identity — getauid, 99
  set user audit identity — setauid, 99
user ID, set real and effective — setreuid(), 242
user IDs
  get — getuid, geteuid, 115
  set — setuid, 245
utime — set file access and modification times, 307
utimes — set file access and modification times, 309

Y
vfork — spawn new process in a virtual memory efficient way, 311
vhangup — the current controlling terminal, 313

W
wait — wait for child process to stop or terminate, 314
wait on a condition variable —
  _lwp_cond_reltimedwait, 126
wait on a condition variable —
  _lwp_cond_timedwait, 126
wait on a condition variable —
  _lwp_cond_wait, 126
wait for a LWP to terminate — _lwp_wait, 140
waitid — wait for child process to change state, 316
waitpid — wait for child process to change state, 318
wracct — get, put, or write extended accounting data, 95
write on a file
  — write, 320
write — write on a file, 320

yield — yield execution to another lightweight process, 326
yield execution to another lightweight process — yield, 326