Trusted Solaris 2.5 Man Pages: 2TSOL System Calls

Sun Microsystems Federal, Inc.
A Sun Microsystems, Inc. Business
901 San Antonio Road, MS USJC01-201
Palo Alto, CA 94303
U.S.A.

Preface

In the Trusted Solaris Reference Manual, each collection of information on a particluar topic is called a man page, even though a man *page* may actually consist of *many pages* of text.

A man page is intended to answer concisely the question “What does it do?”. The man pages are not intended to be a tutorial. Depending what you are trying to do, refer to the other Trusted Solaris user, developer, and administrator manuals for when and why to use a command or other features described in the man pages.

ACCESSING MAN PAGES

The man pages that make up the reference manual may be accessed in three ways.

Note: The following discussion of man page viewing options uses the term **package**, which is a unit of software that is typically delivered on Sun’s product CDs. Installing the documentation packages is optional, because they are not required for operations. Each customer’s administrators decides whether or not the documentation packages are installed and made available.

The first means of accessing the man pages is through the use of the **man**(1) command. When the contents of the man page package, SUNWman, are available on the local system, anyone with a login account, plus a terminal emulator (such as **cmdtool**(1), **shelltool**(1), or **dtterm**(1)) and the **man**(1) command in one of the account’s execution profiles can view a man page on-line. (For more about Trusted Solaris execution profiles and user accounts, see the Trusted Solaris user and administrator documentation.) To view a man page, enter the **man** command followed by the name of the man page. For example, to view the **ls**(1) man page that describes the command used to print out a directory's contents, a user enters the command: **man**ls.

The second way to read man pages is in the printed Trusted Solaris Reference Manual. The reference manual is in the Trusted Solaris documentation set, and it may be ordered in hardcopy form from Sun by using part number: 805-8005-10.

The third means of reading the man pages is by viewing them in AnswerBook format. When the Trusted Solaris AnswerBook package, SUNWtab, is available on the local system, anyone with a login account and with the **answerbook**() command and a terminal emulator in an execution profile can display the Trusted Solaris reference manual and the other user documentation. For Trusted Solaris 2.5, the Trusted Solaris documentation AnswerBook is shipped on a separate documentation CD, but it may be bundled on the same CD with the Trusted Solaris software in future releases.

Trusted Solaris man pages are identified with a TSOL suffix in the section name. The TSOL suffix is used for man pages that are either new to Trusted Solaris or modified from the base man pages from the Solaris, CDE, or Solstice products that are bundled into Trusted Solaris. The man pages are organized alphabetically by section.

- Section 1TSOL describes new or modified user commands available with the Trusted Solaris operating system.

- Section 1BTSOL describes printer commands adapted for Trusted Solaris from the Berkeley Software Distribution (BSD) print subsystem, which are used chiefly for printing administration.

  Note: Use of the equivalent System V print commands is recommended (such as **lp**(1TSOL)instead of **lpr**(1BTSOL)) because although the BSD commands are included for compatability, they will be removed in future releases.

- Section 1MTSOL describes Trusted Solaris system maintenance and administration commands.

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

- 3*TSOL subsections describe functions found in various Trusted Solaris libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2TSOL.

Subsections include: 3CTSOL, 3NTSOL, 3RTSOL, 3TSOL, and 3X11TSOL.

- Section 4TSOL outlines the formats of various files. The C structure declarations for the file formats are given where applicable.
- Section 5TSOL contains miscellaneous documentation such as Trusted Solaris macros.
- 7*TSOL subsections describe various special files that refer to specific hardware peripherals and device drivers.

  Subsections include: 7DTSOL and 7TSOL.
- 9*TSOL subsections provide reference information for writing device drivers in the kernel operating system environment.

  Subsections include: 9FTSOL and 9TSOL.

Following is a generic list of headings on each man page. The man pages of each manual section include only the headings they need. For example, if there are no bugs to report, there is no BUGS section. See the intro pages for more information and detail about each section, and **man**(1) for more information about man pages in general.

NAME

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

SYNOPSIS

This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full pathname is shown. Literal characters (commands and options) are in **bold** font and variables (arguments, parameters and substitution characters) are in *italic* font. Options and arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

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

. . .	Ellipses. Several values may be provided for the previous argument, or the previous argument can be specified multiple times, for example, ‘*filename* . . .’.
|	Separator. Only one of the arguments separated by this character can be specified at time.
{ }	Braces. The options and/or arguments enclosed within braces are interdependent, such that everything enclosed must be treated as a unit.

PROTOCOL

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

AVAILABILITY

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

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

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

MT-LEVEL

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

DESCRIPTION

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

IOCTL

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

OPTIONS

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

OPERANDS

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

OUTPUT

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

RETURN VALUES

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

ERRORS

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

USAGE

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

Commands
Modifiers
Variables
Expressions
Input Grammar

EXAMPLES

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

example%

or if the user must be in an administrative role,

example#

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

ENVIRONMENT

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

EXIT STATUS

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

FILES

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

SEE ALSO

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

DIAGNOSTICS

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

WARNINGS

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

NOTES

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

BUGS

This section describes known bugs and wherever possible suggests workarounds.

SUMMARY OF TRUSTED SOLARIS CHANGES

On base man pages that have Trusted Solaris modifications, this section summarizes the changes in a single easy-to-find place on the man page.

NAME

Intro, intro – introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

Section 2 of the *Trusted Solaris Reference Manual* describes, in alphabetical order, all the system calls available with the Trusted Solaris operating system, which is based on the Solaris operating system, the Common Desktop Environment (CDE) window system, and the Solstice AdminSuite set of system administration tools. Man pages whose section IDs end with the 2TSOL suffix describe system calls that are either new or modified to work within Trusted Solaris security policy. An example of a new Trusted Solaris system call is **secconf**, which is described on the **secconf**(2TSOL) man page. The **secconf** system call allows processes to determine the value of a configurable security-related system variable, such as the variable that hides upgraded file names when set.

Modified system calls are system calls from any of the base products that have been modified to work within the Trusted Solaris *security policy*, such as: **link**. Man pages for modified system calls have been rewritten to remove information that is not accurate for how the system call behaves within the Trusted Solaris system. Modified man pages, such as **link**(2TSOL), also add descriptions for any new features and arguments added to the base.

When a man page for a system call states that the calling process must have or must assert a specified *privilege* or privileges, that means:

- The privilege(s) must be made available as *allowed* privileges on the executable, and
- The privileges must be made available to the effective privilege set of the process in either of these two ways:
  - By inheritance from the parent process, or
  - As *forced* privileges assigned to the executable program.

See *Process Privilege Sets* and *Inheritable Privileges* in the **DEFINITIONS**, and see also the *Trusted Solaris Developer's Guide* for more complete descriptions of the topics mentioned here.

NOTE

The *printed* version of the *Trusted Solaris Reference Manual* includes only the Trusted Solaris man pages, while the *on-line man pages* that are viewable with the **man**(1) command include all the base man pages along with the Trusted Solaris man pages. Printed versions of the unmodified base Solaris man pages are in the Solaris reference manual. The **man** command without any options always displays the Trusted Solaris version, so when both a base man page and a Trusted Solaris version exist, if you want to view the original man page you must use the **man** command with the **–s** option to specify the base section ID of the man page. For example, to display the **link**(2) man page instead of the modified **link**(2TSOL) man page, you would enter: **man –s2 link.** To find out all the sections that contain man pages with the same name, enter: **man –l <man_page_name>.**

ERRORS

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

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

Applications should use bound threads rather than the **_lwp_**∗ system calls (see **thr_create**(3T)). Using LWPs 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>**.

1 **EPERM** Appropriate privilege not asserted
Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or a process with the appropriate *privilege*. It is also returned for attempts by ordinary users to do things that always require a privilege. See *Privilege* in the **DEFINITIONS**.

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

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

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

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

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

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

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

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

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

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

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

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

13 **EACCES** Permission denied
An attempt was made to access a file in a way forbidden by the Trusted Solaris *security policy*. This type of failure due to DAC or MAC restrictions may be bypassed at the discretion of the security administrator if the appropriate override *privilege*(s) are made available to be asserted by the calling process (which privilege to use depending on the type of access being denied). See *Discretionary Access Control*, *File Access*, *Mandatory Access Control*, *Privilege*, and *Security Policy* in the **DEFINITIONS.**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

37 **ECHRNG** Channel number out of range

38 **EL2NSYNC** Level 2 not synchronized

39 **EL3HLT** Level 3 halted

40 **EL3RST** Level 3 reset

41 **ELNRNG** Link number out of range

42 **EUNATCH** Protocol driver not attached

43 **ENOCSI** No CSI structure available

44 **EL2HLT** Level 2 halted

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

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

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

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

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

58–59 Reserved

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

61 **ENODATA** No data available

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

82 **EREMCHG** Remote address changed

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

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

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

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

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

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

89 **ENOSYS** Operation not applicable

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

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

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

93 **ENOTEMPTY** Directory not empty

94 **EUSERS** Too many users

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

151 **ESTALE** Stale NFS file handle

DEFINITIONS

ACL
See *Access Control List*

ACL Mask
Created for compatability purposes, it masks out any existing ACL entries without destroying them when **chmod**(1) changes permissions on a file or directory. The masked names can then later be restored if **chmod** is run again to restore the original permissions.

Access Control List
A type of *discretionary access control* based on a list of entries that the owner can specify for a file or directory. The access control list (ACL) restricts or permits access to any number of individuals and groups, allowing finer-grained control than provided by the standard UNIX *permission bits*.

Accreditation Range
Actually not a range, but a set made up of labels. See *user accreditation range* and *system accreditation range* for more about the two types of accreditation ranges in the Trusted Solaris system.

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.

CMW Label A structure that holds both an *information label* and a *sensitivity label*, this construct allows the information label and sensitivity label to be programmatically translated and manipulated either as single entities or as a combined unit.

Classification The hierarchical portion of a *sensitivity label*, *information label*, or *clearance*, each of which has only one classification. Each classification has an external name (text string) and an internal number (integer), with the lowest number assigned to the lowest classification and the other numbers assigned to the rest of the classifications in a hierarchical relationship. In a sensitivity label assigned to a file or directory, a classification indicates a relative level of protection based on the sensitivity of the information contained in the file or directory. In a clearance assigned to a user and that user's processes, a classification indicates a level of trust.

Clearance An upper bound on the set of labels whose lower bound is the *minimum label* assigned by the security administrator as the *initial label*. Like a *sensitivity label*, a clearance consists of a *classification* and a set of *compartments*. See *Process Clearance*.

Compartment A word associated with one or more compartment bits that may be defined in the **label_encodings**(4TSOL) file to be part of a *sensitivity label*, *information label*, or *clearance.* Compartments represent areas of interest or work groups associated with the labels that contain compartments and are used in MAC decisions. Compartments have no intrinsic ordering; however, the **label_encodings** file can impose constraints that may be hierarchical on the allowable combinations of compartments with each other and with *classifications*.

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

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

DAC See *discretionary access control.*

Device Objects Device objects include printers, workstations, tape drives, floppy drives, audio devices, and internal pseudo terminal devices. See *mandatory acces control* for definitions of MAC policy. Devices are subject to the read equal write equal policy.

Directory Directories organize files into a hierarchical system where directories are the nodes in the hierarchy. A directory is a file that catalogues the list of files, including directories (subdirectories), that are directly beneath it in the hierarchy. Entries in a directory file are called links. A link associates a file identifier with a filename. By convention, a directory contains at least two links, **.** (dot) and **..** (dot-dot). The link called dot refers to the directory itself while dot-dot refers to its parent directory. The root directory, which is the top-most node of the hierarchy, has itself as its parent directory. The pathname of the root directory is **/** and the parent directory of the root directory is **/**.

Discretionary Access Control
The type of access granted or denied by the owner of a file or directory at the discretion of the owner. The Trusted Solaris system provides two kinds of discretionary access (DAC) controls, *permission bits* and *access control lists*.

Disjoint
When two labels of any type (*sensitivity label*, *information label*, or *clearance*) are compared and neither of the two labels *dominate*s the other, the labels are said to be disjoint. Information flow between disjoint labels is considered to be a downgrade.

Dominate
When any type of label (*sensitivity label*, *information label*, or *clearance*) has a security level equal to or greater than the security level of another label to which it is being compared, the first label is said to dominate the second. The classification of the dominant label must equal or be higher than the classification of the second label, and the dominant label must include all the words (compartments and markings, if present) in the other label. Sensitivity labels are compared for dominance when MAC decisions are being made. See *strictly dominate* and *disjoint*.

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

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

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

File Access
Even though, strictly speaking, files, directories, devices and other objects are treated as files in the UNIX system, only the access rules for *file system objects* are described in this section. Because files, directories, and devices have slightly different mandatory access rules, these rules are separately described. See *process objects*, *System V IPC objects*, *STREAMS objects*, *network endpoint objects*, *device objects*, *and X window objects* for the rules that apply to these other types of objects.

A file, directory, or device may be accessed in three ways:

- The *name* of the file, directory, or device may be *viewed*,
- The *contents* or the *attributes* of the file, directory, or device may be *viewed*, or
- The *contents* or the *attributes* of the file, directory, or device may be *modified*.

In the Trusted Solaris system, each of these types of access is granted or denied based on whether certain *discretionary access control* checks (described in *File Access Permissions*) and *mandatory access control* checks have been passed.

All types of access require that the *sensitivity label* of the process *dominate*s the sensitivity label of all directories in the path prefix and that the owner of the process has discretionary access for each directory in the path prefix. View access to the name of the file, directory, or device requires only that this part of the check is passed (unless the system is configured to hide upgraded names).

For view access (*read* access) to the contents or attributes of a file or directory, the process' sensitivity label must dominate the sensitivity label of the file or directory. For view access to the contents of a device (for example, so you can read information on a tape in a tape drive), the process' sensitivity label must be equal to the sensitivity label of the device. The owner of the process also must have discretionary read access to the file, directory, or device.

For a process to write a file or to modify its attributes, the sensitivity label of the file must dominate the sensitivity label of the process and must be dominated by the process' clearance. (See *process clearance*.) For a process to write into a directory (to *create* a file or a symbolic link) the label of the process must equal the sensitivity label of the directory. For a process to write to a device (for example, store information on a tape in a tape drive), the sensitivity label of the process must equal the sensitivity label of the device. The owner of the process must have discretionary write access to the file, directory, or device.

For each type of failure of a MAC or DAC check, a specific override *privilege* may be asserted by the process, depending on the type of access being denied. See *process privilege sets*, and *inheritable privileges*.

These conditions and the listed override privileges apply to any type of access:

- If the sensitivity label of the process does not dominate the sensitivity label of a directory in the path prefix, then the process must assert the privilege to search up (search a directory whose sensitivity label dominates the process' sensitivity label), which is *file_mac_search*.
- If the user on whose behalf the process is being executed does not assert discretionary search permission for a directory in the path prefix, then the process must have the privilege to override DAC search restrictions when accessing a directory, which is *file_dac_search*.

These conditions and the listed override privileges apply to view (read) access to a file or directory or to its attributes:

- If the sensitivity label of the process does not dominate the sensitivity label of the file or directory, then the process must assert the privilege to override MAC read restrictions, which is *file_mac_read*.
- If the user on whose behalf the process is being executed does not have discretionary read permission for the file or directory, then the process must assert the privilege to override DAC read restrictions, which is *file_dac_read*.

These conditions and the listed override privileges apply to modify (write) access to a file or directory or to its attributes:

- If the sensitivity label of the file does not dominate or if the sensitivity of the directory or device does not equal the sensitivity label of the process, and if the sensitivity lable of the file, directory, or device is not dominated by the process' clearance, the process must assert the privilege that overrides MAC write restrictions, allowing the user to write up and to write above the process' clearance, which is *file_mac_write*.
- If the user on whose behalf the process is being executed does not have discretionary write permission for the file or directory, then the process must assert the privilege to override DAC write restrictions, which is *file_dac_write*.

File Access Permissions

Read, write, and execute or search permissions on a file or directory are granted to a process if the *mandatory access control* checks are passed as described in *File Access*, and if one of the following tests is true.

If an ACL exists for a file or directory, then the following tests are performed in order until one is true, and then the requested access is granted.

- If the effective UID of the process is equal to the UID of the owner of the file or directory and if the ACL grants the desired type of access to the owner.
- If the effective UID of the process is in the user list in the ACL and if both the ACL for the owner and the ACL mask grant the desired type of access to the named user.
- If the effective GID or a supplementary GID of the owner of process is equal to the GID of the file or directory and if both the ACL entry for the owner's group and the ACL mask both grant the desired type of access to the owner's group.
- If the effective GID or a supplementary GID of the owner of the process is named in the ACL group list, and if both the ACL entry for the named group and the ACL mask grant the desired type of acess to the group.
- If the ACL's "other" entry grants the desired type of access to the owner of the process.

If an ACL does not exist for the file or directory being accessed, then the following tests are performed in order, and if one of them is passed the desired access is granted.

- If the effective UID of the process is equal to the UID of the owner of the file or directory and if the owner portion (0700) of the file's permissions is set to allow the desired type of access.
- If the effective GID is equal to the GID of the file or directory and if the group portion (0070) of the file's permissions is set to allow the desired type of access.
- If one of the groups in the supplementary group list of the process is equal to the GID of the file or directory and if the group portion (0070) of the file's permissions is set to allow the desired type of access.
- If the other portion (0007) of the file's permission bits is set to allow the desired type of access to all others.

Otherwise, access is denied, unless the process asserts the appropriate DAC override *privilege*(s) described in *File Access*.

File Descriptor

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

File Name

Names consisting of 1 to **NAME_MAX** characters may be used to name an ordinary file, special file or directory.

These characters may be selected from the set of all character values excluding \0 (null) and the ASCII code for **/** (slash).

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

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

File Privilege Sets

These sets consist of the *allowed* and *forced* privileges specified for use by executable files (programs). The allowed set limits which privileges a process can use, whether the privileges are forced on the executable file or inherited (see *inheritable privileges*). Any privileges in the forced privilege set are available to any process that invokes the program, as long as they are also in the allowed set.

File System Objects

File-system objects include files (regular files, process files, and device-special files), directories, symbolic links, FIFOs (named pipes), pipes, and UNIX domain socket rendezvous. See *mandatory access control* for definitions of the MAC policies. See *File Access* for the MAC rules that apply to regular files, device files, symbolic links, and directories. The policies for the remaining file objects are as follows. UNIX domain socket rendezvous and FIFOs (named pipes) are subject to the write up read down policy. Pipes are subject to the read equal write equal policy.

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.

Information Label

A *label* that signifies the present actual sensitivity level of the information with which it is associated while also representing any required markings or handling caveats that apply to that information. An information label consists of a hieracherical *classification* and a set of non-hierarcherical *compartments* that together make up the *information level* of the information label, along with a set of non-hierarchical *markings*.

Information Label Floating
A conjoining of two *information labels* that occurs when information from an object with one information label flows to an object or process that has another information label, the resulting information label reflects the combined *security level* and markings from both information labels, according to the rules defined for valid labels in the **label_encodings**(4TSOL) file.

Inheritable Privileges
The *privilege*s that a process can pass to a program across an **execve**(2V) without their being affected by the new program's *forced* or *allowed* privilege sets. [A child process created through a **fork**(2) receives all of a parent process‘ privilege sets with no change.] When a new program is executed by a process, the inheritable set of the process is set to be equal to the inheritable set of the old program: I[process]=I[program]. The inheritable set is not affected by the forced or allowed privileges on the currently executing program, which allows allows privileges to be passed from programs that cannot use them to programs that can.

{IOV_MAX}
Maximum number of entries in a **struct iovec** array.

Label
A security identifier assigned to an object based on the level of protection it needs and to a process based on the degree of trust afforded to the user on whose behalf the process is running.

Label Range
A set of sensitivity *label*s assigned to allocatable devices, commands and file systems, specified by designating a maximum label and a minimum label. For allocatable devices, the minimum and maximum labels limit the sensitivity labels at which devices may be allocated. [See **allocate**(1TSOL) For commands the minimum and maximum labels limit the sensitivity labels at at which the command may be executed. For file systems, the minimum and maximum labels limit the sensitivity labels at which information may be stored on each file system.

Label View Flags
These flags control the translation and display of the internal ADMIN_LOW and ADMIN_HIGH labels. If no value is set on the Label View Flag, the Default Label View specified in the **label_encodings**(4TSOL) file is used. A value of **External** specifies that the actual label ADMIN_LOW displays as the lowest label name in the user accreditation range specified in the **label_encodings** file, and that the actual label ADMIN_HIGH displays as the highest label name in the user accreditation range. A value of **Internal** specifies that the ADMIN_LOW and ADMIN_HIGH labels are translated to the *Admin Low Name* and *Admin High Name* strings specified in the **label_encodings** file. If no such names are specified, the strings “ ADMIN_LOW” and “ ADMIN_HIGH” are used.

Label Translation Flags
These fifteen-bit flags support the GFI FLAGS= option in the **label_encodings**(4TSOL) file, which allows the use of these flags by applications written to use them. These flags are viewable and modifiable only by a trusted path process.

{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**.

MAC See *mandatory access control.*

MLD See *multilevel directory*.

Mandatory Access Control A type of control based on comparing the *sensitivity label* of an *object* to the sensitivity label of the process that is trying to access the object. The MAC policies that apply to various types of objects are *read equal*, *write equal*, *read down*, and *write up*. (See the individual definitions for each object type for the policy that applies.) When the *read equal* policy applies, an object may be accessed for reading only when the sensitivity label of the process is equal to the sensitivity label of the object. When the *write equal* policy applies, an object may be accessed for writing only when the sensitivity label of the process is equal to the sensitivity label of the object. When the *write up.* policy applies, an object may be accessed for writing only when the sensitivity label of the process is dominated by the sensitivity label of the object, hence the process "writes up" to the object. The write up policy also includes write equal. When the *read down* policy applies, an object may be accessed for reading only when the sensitivity label of the process dominates the sensitivity label of the object, hence the process "reads down" to the object. The read down policy also includes read equal.

Marking A codeword, handling caveat, control or release marking word or set of words, and the associated marking bits, that are used only in *information labels*. Markings have no intrinsic ordering, but **label_encodings**(4TSOL) can impose constraints that may be hierarchical on the allowable combinations of markings and compartments and classifications.

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

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

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

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

```
struct      ipc_perm msg_perm;
struct      msg *msg_first;
struct      msg *msg_last;
```

```
ulong       msg_cbytes;
ulong       msg_qnum;
ulong       msg_qbytes;
pid_t       msg_lspid;
pid_t       msg_lrpid;
time_t      msg_stime;
time_t      msg_rtime;
time_t      msg_ctime;
```

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

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

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

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

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

msg_cbytes is the current number of bytes on the queue.

msg_qnum is the number of messages currently on the queue.

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

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

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

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

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

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

Message Operation Permissions

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

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

Read and write permissions on a **msqid** are granted to a process if the read-equal write-equal *mandatory access control* check is passed or if the process asserts the appropriate override privilege (either PRIV_IPC_MAC_READ or PRIV_IPC_MAC_WRITE), and if one of the following tests is true.

If a System V IPC ACL exists for the message queue, then the following tests are performed in order until one is true, and then the requested access is granted.

- If the effective UID of the process is equal to the UID of the owner of message queue and if the ACL grants the desired type of access to the owner.
- If the effective UID of the process is in the user list in the ACL and if both the ACL for the owner and the ACL mask grant the desired type of access to the named user.
- If the effective GID or a supplementary GID of the owner of process is equal to the GID of the message queue and if both the ACL entry for the owner's group and the ACL mask both grant the desired type of access to the owner's group.
- If the effective GID or a supplementary GID of the owner of the process is named in the ACL group list, and if both the ACL entry for the named group and the ACL mask grant the desired type of acess to the group.
- If the ACL's "other" entry grants the desired type of access to the owner of the process.
- The process has asserted the appropriate DAC privilege, PRIV_FILE_DAC_READ or PRIV_FILE_DAC_WRITE.

If a System V IPC ACL does not exist for the message queue being accessed, then the following tests are performed in order, and if one of them is passed the desired access is granted.

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

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

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

The process has asserted the appropriate DAC privilege, either PRIV_FILE_DAC_READ or PRIV_FILE_DAC_WRITE.

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.

Multilevel Directory

A directory in which information at differing *sensitivity labels* is maintained in separate subdirectories called *single-level directories* (SLDs), while appearing to most interfaces to be a single directory under a single name. In the Trusted Solaris system, directories that are used by multiple standard applications to store files at varying labels, such as the **/tmp** directory, **/var/spool/mail** , and users' $HOME directories are set up to be MLDs. A process can access an MLD two ways: either by using pathname translation, and using the adorned name. When a process refers to an MLD without the adorned name, Trusted Solaris transparently extends the reference to the SLD that corresponds to the process' sensitivity label. If the process is creating a file and if the correct SLD does not already exist, Trusted Solaris creates the SLD and assigns it the process' sensitivity label so that the correct single-level directory exists for the file. If the process wants to access the MLD directly, it should use the the MLD *adornment* on the final component of the path. The text string .MLD. is the default adornment. The adornment is a file system attribute that may be changed using **setfsattr**(1MTSOL). Use of the adornment allows programs to refer directly to the MLD instead of to the SLD that has the same SL as the process.

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.

Network Endpoint Objects

Network endpoint objects are sockets and the transport level interface (TLI). See *mandatory access control* for definitions of the MAC policies. Network endpoint objects are subject to the read equal write equal policy.

Object

Anything in the Trusted Solaris system that a process attempts to access. The six major object types are *file system objects*, *process objects*, *System V IPC objects*, *STREAMS objects*, *network endpoint objects*, *device objects*, and *X window objects*.

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

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

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

A slash by itself names the root directory.

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

Permission Bits
A type of *discretionary access control* in which the owner specifies a set of bits to signify who can read, write, or execute a file or directory. Three different sets of permissions are assigned to each file or directory: one set for the owner; one set for all members of the group specified for the file or directory; and one set for all others. See also *access control lists.*

Privilege
Having appropriate privilege means having the capability to override some aspect of *security policy*. If a man page states that a system call needs to have or to assert "the appropriate privilege" to bypass DAC or MAC restrictions, see *File Access Permissions* for the override privilege that applies to the type of access being denied. A privilege is only granted by a site's *security administrator* after judging that the command itself or the person will use the privilege in a trustworthy manner. See *File Privilege Sets and Process Privilege Sets.*

Privilege Debugging Flag
This one bit flag indicates that the process is in privilege debugging mode, an operational mode where any attempt by the process to use a privilege is logged. This flag can be viewed or cleared, but can be set only by a trusted path process. This flag is set by **runpd**(1MTSOL) when executing a command in privilege debugging mode, and then is inherited by the process. It works only if the TSOL_PRIVS_DEBUG kernel switch is also enabled (see **secconf**(2TSOL).

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

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

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

Process Attribute Flags Trusted Solaris flags that indicate security-related values that are copied from one process to another on **fork**(2TSOL) and cloned without changes on **exec**(2TSOL). They are: the *Trusted Path Flag*, the *Privilege Debugging Flag*, the *Network Token Mapping Process Flag*, the *Label View Flag* (External View or Internal View), the *Label Translation Flags*, the *Part of Diskless Boot Flag* and the *Part of Cut and Past Selection Agent Flag*. See **pattr**(1TSOL), **getpattr**(2TSOL)C , and **setpattr**(2TSOL). Each flag has its own protection policy. Any process may view or clear any process attributes flags except for the Label Translation flags, which are viewable and clearable by only a process with the trusted path attribute. Any process may set the Label View flags, but only processes with the trusted path attribute may set any of the other process attribute flags.

Process Clearance At the time a process is created, the process clearance is inherited from the parent process. A process cannot change its own *sensitivity label* or that of a file to a sensitivity label that strictly dominates the process clearance and cannot access (or write up to) a file or other process whose sensitivity label strictly dominates its clearance, without having and using *privileges*.

Process Group Each process in the system is a member of a process group that is identified by a process group ID. Any process that is not a process group leader may create a new process group and become its leader. Any process that is not a process group leader may join an existing process group that shares the same session as the process. A newly created process joins the process group of its parent.

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

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

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

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

Process Objects Process and lightweight processes (independently scheduled threads of execution), which are subject to the write up read down policy. See *object* for definitions of the MAC policies.

Process Privilege Sets The privileges used by a process are stored in sets called the *inheritable*, *permitted*, *effective*, and *saved* sets. When a process executes a program through the **execve**(2TSOL) system call, the permitted (P) and effective (E) privilege sets are reset equal to the same value, which is the intersection of the process' previously existing inheritable (I) privileges and the program file's allowed (A) privileges intersected with the program file's forced (F) privileges: P=E=(I[process] union F[program] restricted by A[program]). The saved privilege set is set initially to the intersection of the existing inheritable privilege set and the file's allowed privileges: S=(I[process] intersected by A), which allows the process to determine which privileges it had when the currently executing program was invoked. When a new program is invoked, the inheritable privilege set is initially set to be the same as the inheritable privileges of the process that invoked the current program: I[new]=I[old]. Setting the inheritable privileges without reference to the forced or allowed privileges on an executing program allows privileges to be passed without change from a program that cannot use them to one that can. For compatability with the base system's super-user capability, if the effective UID is set by **setuid**(2TSOL), to be different from the original, the effective set is copied to the saved set and the effective set is cleared: S=E; E=0. When the process changes its effective user ID back to the original, the saved privilege set is copied to the effective set, thus restoring its privileged state: E=S. In addition to automatic changes in privilege sets as the result of **execve** or **setuid**, a process may manipulate its own privilege sets with the **getppriv**(2TSOL) and **setppriv**(2TSOL) system calls. For example, a process can use these calls to move permitted privileges into and out of its effective privilege set, for privilege bracketing. A process with the PRIV_SET_FPRIV privilege in its effective set can use **setfpriv**(2TSOL) to set privileges on a file. See the *Trusted Solaris Developer's Guide* for more details about how privileges may be manipulated within programs using system calls.

Process Security Attribute

Security attributes received by processes from the base Solaris system are: the process ID (PID), the real, effective, and saved user ID, the real, effective, or saved ID, the supplementary group IDs, the user audit ID, the audit session ID, the audit preselection mask, the terminal ID, and the **umask**(see **Masks**). Security attributes received by processes from the Trusted Solaris system are: the process *clearance*, the *CMW label*, the *process attribute flags*, and the permitted, effective, inheritable, and saved *process privilege sets*.

Read Queue

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

Real User ID and Real Group ID

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

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

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

Root Directory and Current Working Directory

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

Saved User ID and Saved Group ID

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

SLD See *single-level directory*.

Security Attribute An attribute used in enforcing the Trusted Solaris security policy. Various sets of security attributes, both from the base Solaris and the Trusted Solaris systems, are assigned to processes, users, files, directories, file systems, hosts on the trusted network, allocatable devices, and other entities. See *Process Security Attributes.*

Security Policy In the Trusted Solaris environment, the set of rules for DAC, MAC, information labeling, and privilege interpretation that define how information may be accessed. At a customer site, the set of rules that define the sensitivity of the information being processed at that site and the measures that are used to protect the information from unauthorized access.

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

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

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

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

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

sem_nsems is equal to the number of semaphores in the set. Each semaphore in the set is referenced by a nonnegative integer referred to as a **sem_num**. **sem_num** values run sequentially from 0 to the value of **sem_nsems** minus 1.

sem_otime is the time of the last **semop** operation.

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

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

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

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

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

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

semzcnt is a count of the number of processes that are currently suspended awaiting this semaphore's semval to become 0.

Semaphore Operation Permissions

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

00400	**READ by user**
00200	**ALTER by user**
00040	**READ by group**
00020	**ALTER by group**
00004	**READ by others**
00002	**ALTER by others**

Read and alter permissions on a semid are granted to a process if the read-equal write-equal *mandatory access control* check is passed or if the process asserts the appropriate override privilege (either PRIV_IPC_MAC_READ or PRIV_IPC_MAC_WRITE), and if one of the following tests is true.

If a System V IPC ACL exists for the semaphore, and then the following tests are performed in order until one is true, and then the requested access is granted.

- If the effective UID of the process is equal to the UID of the owner of semaphore and if the ACL grants the desired type of access to the owner.
- If the effective UID of the process is in the user list in the ACL and if both the ACL for the owner and the ACL mask grant the desired type of access to the named user.
- If the effective GID or a supplementary GID of the owner of process is equal to the GID of the semaphore and if both the ACL entry for the owner's group and the ACL mask both grant the desired type of access to the owner's group.
- If the effective GID or a supplementary GID of the owner of the process is named in the ACL group list, and if both the ACL entry for the named group and the ACL mask grant the desired type of acess to the group.
- If the ACL's "other" entry grants the desired type of access to the owner of the process.
- The process has asserted the appropriate DAC privilege, PRIV_FILE_DAC_READ or PRIV_FILE_DAC_WRITE.

If a System V IPC ACL does not exist for the semaphore being accessed, then the following tests are performed in order, and if one of them is passed the desired access is granted.

> 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 or the supplementary 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.
>
> The process has asserted the appropriate DAC privilege, either PRIV_FILE_DAC_READ or PRIV_FILE_DAC_WRITE.

Otherwise, the corresponding permissions are denied.

Sensitivity Label A *label* that defines the level of protection afforded to an *object* or or the level of access granted a process, it consists of a hieracherical *classification* and a set of non-hierarcherical *compartments*, which together make up the level of the sensitivity label. The sensitivity labels on objects and processes are compared for dominance (see *Dominate*) in all MAC decisions by the Trusted Solaris system.

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

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

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

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

Shared Memory Identifier

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

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

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

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

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

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

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

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

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

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

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

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

Shared Memory Operation Permissions

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

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

Read and write permissions on a **shmid** are granted to a process if the read-equal write-equal *mandatory access control* check is passed or if the process asserts the appropriate override privilege (either PRIV_IPC_MAC_READ or PRIV_IPC_MAC_WRITE), and if one of the following tests is true.

If a System V IPC ACL exists for the shared memory, then the following tests are performed in order until one is true, and then the requested access is granted.

- If the effective UID of the process is equal to the UID of the owner of shared memory and if the ACL grants the desired type of access to the owner.
- If the effective UID of the process is in the user list in the ACL and if both the ACL for the owner and the ACL mask grant the desired type of access to the named user.
- If the effective GID or a supplementary GID of the owner of process is equal to the GID of the shared memory and if both the ACL entry for the owner's group and the ACL mask both grant the desired type of access to the owner's group.
- If the effective GID or a supplementary GID of the owner of the process is named in the ACL group list, and if both the ACL entry for the named group and the ACL mask grant the desired type of acess to the group.
- If the ACL's "other" entry grants the desired type of access to the owner of the process.
- The process has asserted the appropriate DAC privilege, PRIV_FILE_DAC_READ or PRIV_FILE_DAC_WRITE.

If a System V IPC ACL does not exist for the shared memory being accessed, then the following tests are performed in order, and if one of them is passed the desired access is granted.

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

The effective group ID or the supplementary 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.

The process has asserted the appropriate DAC privilege, either PRIV_FILE_DAC_READ or PRIV_FILE_DAC_WRITE.

Otherwise, the corresponding permissions are denied.

Single-Level Directory

A directory within an MLD containing only files at a single *sensitivity label*. The SLD directory name is derived from the of the process that created it. For example, the name of an SLD in **/tmp** would be in the form **/tmp/.SLD.**<*sensitivity_label_of_creating_process*>/*file.* All subsequent references to the file in the **/tmp** directory would be made transparently as **/tmp/***file*: because pathname translation is transparent, the process would not need to explicitly reference the SLD directory, unless it chose to do so using the adornment and the name of the SLD.

Special Processes

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

STREAMS

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

Stream

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

Stream Head

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

STREAMS Objects

The only type of STREAMS object is a STREAM, which is subject to the read equal write equal MAC policy. See *Object* for definitions of the MAC policies.

Strictly Dominate

When any type of label (*sensitivity label*, *information label*, or *clearance*) has a security level greater than the security level of another label to which it is being compared, the first label strictly *dominate*s the second label. Strict dominance is dominance without equality, which occurs either when the classification of the first label is higher than that of the second label and the first label contains all the second label's compartments or when the classifications of both labels are the same while the first label contains all the compartments in the second label plus one or more additional compartments.

System V IPC Objects

System V interprocess communication (IPC) *object*s are: *message*queues, *semaphore*s, and *shared*memory.

System Accreditation Range	The set of all valid (well-formed) labels created according to the rules defined by each site's security administrator in the **label_encodings**(4TSOL) file, plus the two administrative labels that are used in every Trusted Solaris system, ADMIN_LOW and ADMIN_HIGH.
Trusted Path Flag	Also called the Trusted Path Attribute, this one-bit flag indicates that the process is executing in the trusted path.
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.
X Window Objects	X window objects are the windows in the common desktop environment (which is based on the X Window system). See *mandatory access control* for definitions of the MAC policies. Window objects are generally subject to the read equal write equal policy. See the X library man pages (in section 3X11TSOL) for exceptions.

Name	**Description**
access(2TSOL)	determine accessibility of a file
acl(2TSOL)	get or set a file's access control list (ACL)
adjtime(2TSOL)	correct the time to allow synchronization of the system clock
audit(2TSOL)	write a record to the audit log
auditon(2TSOL)	manipulate auditing
auditsvc(2TSOL)	write audit log to specified file descriptor
chdir(2TSOL)	change working directory
chmod(2TSOL)	change access permission mode of file
chown(2TSOL)	change owner and group of a file
chroot(2TSOL)	change root directory
chstate(2TSOL)	change the view of a host state between labeled and unlabeled
creat(2TSOL)	create a new file or rewrite an existing one
devpolicy(2TSOL)	get or set device driver policy table
exec(2TSOL)	execute a file
execl(2TSOL)	See **exec**(2TSOL)
execv(2TSOL)	See **exec**(2TSOL)
execle(2TSOL)	See **exec**(2TSOL)
execlp(2TSOL)	See **exec**(2TSOL)
execve(2TSOL)	See **exec**(2TSOL)

execvp(2TSOL)	See **exec**(2TSOL)
facl(2TSOL)	See **acl**(2TSOL)
fchdir(2TSOL)	See **chdir**(2TSOL)
fchmod(2TSOL)	See **chmod**(2TSOL)
fchown(2TSOL)	See **chown**(2TSOL)
fchroot(2TSOL)	See **chroot**(2TSOL)
fcntl(2TSOL)	control files
fgetfattrflag(2TSOL)	See **getfattrflag**(2TSOL)
fgetcmwfsrange(2TSOL)	See **getcmwfsrange**(2TSOL)
fgetcmwlabel(2TSOL)	See **getcmwlabel**(2TSOL)
fgetfpriv(2TSOL)	See **getfpriv**(2TSOL)
fgetfsattr(2TSOL)	See **getfsattr**(2TSOL)
fgetmldadorn(2TSOL)	See **getmldadorn**(2TSOL)
getrlimit(2TSOL)	control maximum system resource consumption
fgetsldname(2TSOL)	See **getsldname**(2TSOL)
fork(2TSOL)	create a new process
fork1(2TSOL)	See **fork**(2TSOL)
fpathconf(2TSOL)	get configurable pathname variables
fsetcmwlabel(2TSOL)	See **setcmwlabel**(2TSOL)
fsetfattrflag(2TSOL)	See **getfattrflag**(2TSOL)
fsetfpriv(2TSOL)	See **getfpriv**(2TSOL)
fstat(2TSOL)	See **stat**(2TSOL)
fstatvfs(2TSOL)	See **statvfs**(2TSOL)
getaudit(2TSOL)	get and set process audit information
getauid(2TSOL)	get and set user audit identity
getclearance(2TSOL)	get process clearance
getcmwfsrange(2TSOL)	get file system sensitivity label range
getcmwlabel(2TSOL)	get a file’s CMW label
getcmwplabel(2TSOL)	get a process’ CMW label
getdents(2TSOL)	read directory entries and put in a file system independent format
getfattrflag(2TSOL)	set or get the security attribute flags for a file
getfpriv(2TSOL)	get or set the privilege set for a file
getfsattr(2TSOL)	get filesystem security attributes
getgroups(2TSOL)	get or set supplementary group access list IDs

getmldadorn(2TSOL)	get file system multilevel directory adornment
getmsgqcmwlabel(2TSOL)	get the CMW labels associated with System V IPC structures
getpattr(2TSOL)	get or set process attribute flags
getpid(2TSOL)	get process, process group, and parent process IDs
getppriv(2TSOL)	return or assign a privilege set associated with the invoking process
getrlimit(2TSOL)	control maximum system resource consumption
getsemcmwlabel(2TSOL)	See **getmsgqcmwlabel**(2TSOL)
getshmcmwlabel(2TSOL)	See **getmsgqcmwlabel**(2TSOL)
getsid(2TSOL)	get or set session ID
getsldname(2TSOL)	get file system single-level directory name
kill(2TSOL)	send a signal to a process or a group of processes
lchown(2TSOL)	See **chown**(2TSOL)
lgetcmwlabel(2TSOL)	See **getcmwlabel**(2TSOL)
link(2TSOL)	link to a file
llseek(2TSOL)	move extended read / write file pointer
lseek(2TSOL)	move read / write file pointer
lsetcmwlabel(2TSOL)	See **getcmwlabel**(2TSOL)
lstat(2TSOL)	See **stat**(2TSOL)
mkdir(2TSOL)	make a directory
mknod(2TSOL)	make a directory, or a special or ordinary file
mldgetfattrflag(2TSOL)	See **getfattrflag**(2TSOL)
mldsetfattrflag(2TSOL)	See **getfattrflag**(2TSOL)
mldstat(2TSOL)	See **stat**(2TSOL)
mldlstat(2TSOL)	See **stat**(2TSOL)
mount(2TSOL)	mount a file system
msgctl(2TSOL)	message-control operations
msgget(2TSOL)	get message queue
msggetl(2TSOL)	See **msgget**(2TSOL)
msgop(2TSOL)	message operations
msgrcvl(2TSOL)	See **msgop**(2TSOL)
msgsnd(2TSOL)	See **msgop**(2TSOL)
msgrcv(2TSOL)	See **msgop**(2TSOL)
msgsndl(2TSOL)	See **msgop**(2TSOL)
nice(2TSOL)	change priority of a process

open(2TSOL)	open for reading or writing
p_online(2TSOL)	change processor online or offline status
priocntl(2TSOL)	control process schedulers
priocntlset(2TSOL)	generalized process scheduler control
pread(2TSOL)	See **read**(2TSOL)
preadl(2TSOL)	See **read**(2TSOL)
processor_bind(2TSOL)	determine type and status of a processor
pwrite(2TSOL)	See **write**(2TSOL)
pwritel(2TSOL)	See **write**(2TSOL)
read(2TSOL)	read from file
readl(2TSOL)	See **read**(2TSOL)
readlink(2TSOL)	read the value of a symbolic link
readv(2TSOL)	See **read**(2TSOL)
readvl(2TSOL)	See **read**(2TSOL)
rename(2TSOL)	change the name of a file
rmdir(2TSOL)	remove a directory
secconf(2TSOL)	get security configuration information
semctl(2TSOL)	semaphore-control operations
semget(2TSOL)	get set of semaphores
semgetl(2TSOL)	See **semget**(2TSOL)
semop(2TSOL)	semaphore operations
semopl(2TSOL)	See **semop**(2TSOL)
setaudit(2TSOL)	See **getaudit**(2TSOL)
setauid(2TSOL)	See **getauid**(2TSOL)
setfattrflag(2TSOL)	See **getattrflag**(2TSOL)
setgroups(2TSOL)	See **getgroups**(2TSOL)
setpattr(2TSOL)	See **getpattr**(2TSOL)
setfpriv(2TSOL)	See **getfpriv**(2TSOL)
setppriv(2TSOL)	See **getppriv**(2TSOL)
setclearance(2TSOL)	set process clearance
setcmwlabel(2TSOL)	set CMW label of a file
setcmwplabel(2TSOL)	set process CMW label
setegid(2TSOL)	See **setuid**(2TSOL)
seteuid(2TSOL)	See **setuid**(2TSOL)
setgid(2TSOL)	See **setuid**(2TSOL)

setregid(2TSOL)	set real and effective group IDs
setreuid(2TSOL)	set real and effective user IDs
setrlimit(2TSOL)	See **getrlimit**(2TSOL)
setuid(2TSOL)	set user and group IDs
shmat(2TSOL)	See **shmop**(2TSOL)
shmctl(2TSOL)	shared-memory control operations
shmdt(2TSOL)	See **shmop**(2TSOL)
shmget(2TSOL)	get shared-memory segment identifier
shmgetl(2TSOL)	See **shmget**(2TSOL)
shmop(2TSOL	shared-memory operations
sigsend(2TSOL)	send a signal to a process or a group of processes
sigsendset(2TSOL)	See **sigsend**(2TSOL)
stat(2TSOL)	get file status
statvfs(2TSOL)	get file system information
stime(2TSOL)	set system time and date
swapctl(2TSOL)	manage swap space
symlink(2TSOL)	make a symbolic link to a file
sysinfo(2TSOL)	get and set system information strings
tokmapper(2TSOL)	manipulate kernel token mapping caches
uadmin(2TSOL)	administrative control
ulimit(2TSOL)	get and set process limits
umount(2TSOL)	unmount a file system
unlink(2TSOL)	remove directory entry
utimes(2TSOL)	set file times
vfork(2TSOL)	spawn new process in a virtual memory efficient way
write(2TSOL)	write on a file
writel(2TSOL)	See **write**(2TSOL)
writev(2TSOL)	See **write**(2TSOL)
writevl(2TSOL)	See **write**(2TSOL)

NAME

access – Determine accessibility of a file

SYNOPSIS

#include <unistd.h>

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

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

access( ) checks the file to which *path* points 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 check allows a setuid process to verify that the user running it would have had permission to access this file. The bit pattern contained in *amode* is constructed by an OR of the access permissions to be checked (R_OK, W_OK, and X_OK) and the existence test (F_OK). These constants are defined in <**unistd.h**>:

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**(2TSOL) for additional information about file-access permission.

The information label of *path* is unchanged. The information label of the calling process is also unchanged.

RETURN VALUES

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

The information label of *path* is not changed by this system call. The information label of the calling process is not changed by this system call.

ERRORS

Access to the file is denied if any of these conditions is true:

EACCES	Search permission is denied on a component of the path prefix. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_SEARCH** and **PRIV_FILE_MAC_SEARCH**.
EACCES	Permission bits of the file mode do not permit the requested access. The calling process does not have mandatory read, write, execute, or search access to the final component in *path*. To override this restriction, the calling process may assert one or more of these privileges depending on the value in *amode*. **PRIV_FILE_MAC_WRITE** , **PRIV_FILE_DAC_WRITE**, **PRIV_FILE_MAC_READ**, **PRIV_FILE_DAC_READ**, **PRIV_FILE_MAC_SEARCH** (in the case of a directory), **PRIV_FILE_DAC_SEARCH**, and **PRIV_FILE_DAC_EXECUTE**.
EFAULT	*path* points to an illegal address.

EINTR	A signal was caught during the **access** function.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	The *path* argument points to an empty string or to the name of a file that does not exist.
ENOLINK	*path* points to a remote machine but the link to that machine is no longer active.
ENOTDIR	A component of the path prefix is not a directory.
EROFS	Write access is requested for a file on a read-only file system.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

The information labels of *path* and of the calling process are unchanged.

Certain uses of this interface may present a covert channel. If a covert channel is exploited, the execution of the process may be delayed. To avoid this delay, the process may assert the **PRIV_PROC_NODELAY** privilege.

SEE ALSO

intro(2TSOL), **chmod**(2TSOL), **stat**(2TSOL)

NAME

acl, facl – Get or set a file's access control list (ACL)

SYNOPSIS

#include <sys/acl.h>

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

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

DESCRIPTION

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

These three values for *cmd* are available:

SETACL *nentries* ACL entries, specified in buffer *aclbufp*, are stored in the file's ACL. This command can be executed only by a process that has an effective user ID equal to the owner of the file. To override this restriction, the calling process may assert the **PRIV_FILE_SETDAC** privilege.

All directories in the path name must be searchable.

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

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

RETURN VALUES

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

The information label of *pathp* or *fildes* and the information label of the calling process remain unchanged.

The audit record has multiple events that represent the requested function. For SETACL, the audit record includes the old and new ACLs.

ERRORS

acl or **facl** will fail if any of these conditions is true:

EACCES The caller does not have search access to a component of the pathname. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_SEARCH** and **PRIV_FILE_MAC_SEARCH**.

cmd is GETACLCNT or GETACL and the calling process does not have mandatory read access. To override this restriction, the calling process may assert the **PRIV_FILE_MAC_READ** privilege.

cmd is SETACL and the calling process does not have mandatory write access. To override this restriction, the calling process may assert the

	PRIV_FILE_MAC_WRITE privilege.
EINVAL	*cmd* is not **GETACL**, **SETACL**, or **GETACLCNT**.
	cmd is **SETACL** and *nentries* is less than three.
	cmd is **SETACL** and the ACL specified in *aclbufp* is not valid.
EIO	A disk I/O error has occurred while storing or retrieving the ACL.
EPERM	*cmd* is **SETACL** and the effective user ID of the caller does not match the owner of the file. To override this restriction, the calling process may assert the **PRIV_FILE_SETDAC** privilege.
ENOENT	A component of the path does not exist.
ENOSPC	*cmd* is **GETACL** and *nentries* is less than the number of entries in the file's ACL.
ENOSPC	*cmd* is **SETACL** and there is insufficient space in the file system to store the ACL.
ENOTDIR	A component of the path specified by *pathp* is not a directory.
	cmd is **SETACL** and an attempt is made to set a default ACL on a file type other than a directory.
ENOSYS	*cmd* is **SETACL** and the file specified by *pathp* resides on a file system that does not support ACLs. **acl** is not supported by this implementation.
EROFS	*cmd* is **SETACL** and the file specified by *pathp* resides on a file system that is mounted read-only.
EFAULT	*pathp* or *aclbufp* points to an illegal address.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access or ownership checks.

The information label of *pathp* or *fildes* and the information label of the calling process remain unchanged.

The audit record has multiple events that represent the requested function. For SETACL the audit record includes the old and new ACLs.

SEE ALSO

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

NAME

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

SYNOPSIS

#include <sys/time.h>

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

DESCRIPTION

adjtime( ) adjusts the system's notion of the current time, as returned by **gettimeofday**(3C), advancing or retarding it by the amount of time specified in the **struct timeval** pointed to by *delta*.

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

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

The calling process must have the **PRIV_SYS_CONFIG** privilege in order to adjust the time of day.

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

RETURN VALUES

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

ERRORS

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

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

SUMMARY OF TRUSTED SOLARIS CHANGES

The calling process must have the **PRIV_SYS_CONFIG** privilege in order to use this system call.

SEE ALSO

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

NAME

audit – write a record to the audit log

SYNOPSIS

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

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

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

AVAILABILITY

Available only on Trusted Solaris systems with the auditing module enabled. (The auditing module is enabled by default in the Trusted Solaris system.) See **bsmconv**(1MTSOL) for more information. **auditwrite**(3TSOL) is the preferred interface for creating audit records under Trusted Solaris 2.x.

DESCRIPTION

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

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.

If the event number is between 2048 and 32767 the calling process must have the privilege **PRIV_PROC_AUDIT_TCB** in its set of effective privileges. If the event number is between 32768 and 65535, the caller must have the privilege **PRIV_PROC_AUDIT_APPL** in its set of effective privileges.

RETURN VALUES

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

ERRORS

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

EFAULT	*record* points outside the process's allocated address space.
EINVAL	The record header token ID is invalid or the length is either less than the header token size or greater than **MAXAUDITDATA**.
EPERM	The process's effective privilege set does not contain the proper privilege for this operation.

SUMMARY TRUSTED SOLARIS CHANGES

See the **DESCRIPTION** section for information about which privileges are needed to use this call when the event number being audited is in the application set or the kernel set.

SEE ALSO

auditd(1MTSOL), **auditon**(2TSOL), **auditsvc**(2TSOL), **getaudit**(2TSOL), **auditwrite**(3TSOL), **audit.log**(4TSOL)

NAME

auditon – Manipulate auditing

SYNOPSIS

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

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

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

DESCRIPTION

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

COMMANDS

These commands are supported:

A_GETCOND	Returns the system audit on / off / disabled condition in the integer long to which *data* points.
A_SETCOND	Sets the system's audit on / off condition to the value in the integer long to which *data* points. If the current state is disabled, the BSM audit module must be enabled by **bsmconv**(1MTSOL) before auditing can be turned on.
A_GETCLASS	Returns the event-to-class mapping for the designated audit event. The *data* argument points to the **au_evclass_map** structure containing the event number. The preselection class mask is returned in the same structure.
A_SETCLASS	Sets the event-class preselection mask for the designated audit event. The *data* argument points to the **au_evclass_map** structure containing the event number and class mask.
A_GETKMASK	Returns the kernel preselection mask in the **au_mask** structure to which *data* points.
A_SETKMASK	Sets the kernel preselection mask. The *data* argument points to the **au_mask** structure containing the class mask.
A_GETPINFO	Returns the audit ID, preselection mask, terminal ID, and audit session ID of the specified process in the **auditpinfo** structure to which *data* points.
A_SETPMASK	Sets the preselection mask of the specified process. The *data* argument points to the **auditpinfo** structure containing the process ID and the preselection mask.
A_SETUMASK	Sets the preselection mask for all processes with the specified audit ID. The *data* argument points to the **auditinfo** structure containing the audit ID and the preselection mask.

A_SETSMASK	Sets the preselection mask for all processes with the specified audit session ID. The *data* argument points to the **auditinfo** structure containing the audit session ID and the preselection mask.
A_GETQCTRL	Returns the kernel audit-queue control parameters. These control the high- and low-water marks of the number of audit records allowed in the audit queue. Another parameter controls the size of the data buffer used by **auditsvc**(2TSOL) to write data to the audit trail. There is also a parameter that specifies a delay before data is written to the audit trail. The audit-queue parameters are returned in the **au_qctrl** structure to which *data* points.
A_SETQCTRL	Sets the kernel audit-queue control parameters. The *data* argument points to the **au_qctrl** structure containing the audit-queue control parameters.
A_GETCWD	Returns the current working directory as kept by the audit subsystem. This directory is a path anchored on the real root, rather than on the active root. The *data* argument points to a buffer into which the path is copied. The *length* argument provides the length of the buffer.
A_GETCAR	Returns the current active root as kept by the audit subsystem. This path may be used to anchor an absolute path for a path token generated by an application. The *data* argument points to a buffer into which the path is copied. The *length* argument provides the length of the buffer.
A_GETSTAT	Returns the system audit statistics in the **audit_stat** structure to which *data* points.
A_SETSTAT	Resets system audit statistics values.
A_GETPOLICY	Returns the audit policy flags in the integer long to which *data* points.
A_SETPOLICY	Sets the audit policy flags to the values in the integer long to which *data* points. The next sections lists and describes policy flags recognized.

A process must have **PRIV_SYS_AUDIT**, **PRIV_PROC_AUDIT_TCB**, or **PRIV_PROC_AUDIT_APPL** in its set of effective privileges in order to successfully execute these commands: **A_GETCOND**, **A_GETCLASS**, **A_GETPINFO**, **A_GETCWD**, **A_GETCAR**, and **A_GETPOLICY**.

A process must have **PRIV_SYS_AUDIT** in its set of effective privileges in order to successfully execute these commands: **A_SETCOND**, **A_SETCLASS**, **A_GETKMASK**, **A_SETKMASK**, **A_SETPMASK**, **A_SETUMASK**, **A_SETSMASK**, **A_GETQCTRL**, **A_SETQCTRL**, **A_GETSTAT**, **A_SETSTAT**, and **A_SETPOLICY**.

Policy Flags

AUDIT_ACL	Include in the audit data an ACL attribute for each object accessed. Note that regardless of policy, if there is no ACL associated with an object, an attribute will not be generated. This information is not included by default.
AUDIT_AHLT	Halt the machine if an asynchronous audit event occurs that cannot be delivered because the audit queue has reached the high-water mark or because there are insufficient resources to construct an audit record. By default, records are dropped and a count is kept of the number of dropped records.
AUDIT_CNT	Do not suspend processes when audit storage is full or inaccessible. The default action is to suspend processes until storage becomes available.
AUDIT_ARGV	Include in the audit record the argument list for the **exec**(2) system call. The default action is not to include this information.
AUDIT_ARGE	Include in the audit record the environment variables for the **execv**(2) system call. The default action is not to include this information.
AUDIT_SEQ	Add a sequence token to each audit record. The default action is not to include this token.
AUDIT_TRAIL	Append a trailer token to each audit record. The default action is not to include this token.
AUDIT_GROUP	Include the supplementary groups list in audit records. The default action is not to include it.
AUDIT_ILABEL	Include ilabels in audit records unless ilabels are not enabled on this system. This information is not included by default.
AUDIT_SLABEL	Include slabels in audit records. The default action is to include slabels in audit records.
AUDIT_PASSWD	Include as part of the audit record any bad authentication data encountered during a login operation. The default action is not to include the password in the audit record.
AUDIT_PATH	Include secondary paths in audit records. Examples of secondary paths are dynamically loaded, shared library modules and the command shell path for executable scripts.
AUDIT_WINDATA_DOWN	Include in an audit record any downgraded data moved between windows. By default, this data is not included.
AUDIT_WINDATA_UP	Include in an audit record any upgraded data moved between windows. By default, this data is not included.

RETURN VALUES

Upon success, **auditon** returns **0**. Upon failure, **auditon** returns –**1** and sets **errno** to indicate the error.

ERRORS

EFAULT	The copy of data to / from the kernel failed.
EINVAL	One of the system call arguments was illegal.
EPERM	The process did not have the appropriate privilege in its effective set.

SUMMARY OF TRUSTED SOLARIS CHANGES

These policy flags have been added in Trusted Solaris: **AUDIT_ACL**, **AUDIT_AHLT**, **AUDIT_ILABEL**, **AUDIT_SLABEL**, **AUDIT_PASSWD**, **AUDIT_WINDATA_DOWN**, and **AUDIT_WINDATA_UP**. The **DESCRIPTION** section explains which privileges are required to use which audit-control commands.

SEE ALSO

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

NOTES

This functionality is active only if the audit module has been enabled. By default, this module is enabled on Trusted Solaris systems. See **bsmconv**(1MTSOL) for more information.

Trusted Solaris 2.x will soon extend the number of audit classes and introduce new but similar structures and programming interfaces.

NAME

auditsvc – Write audit log to specified file descriptor

SYNOPSIS

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

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

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

DESCRIPTION

The **auditsvc** system call specifies the audit log file to the kernel. The kernel writes audit records to this file until an exceptional condition occurs and then the call returns. The parameter *fd* is a file descriptor that identifies the audit file. Programs should open this file for writing before calling **auditsvc**. The parameter *limit* specifies the number of free blocks that must be available in the audit-file system; **auditsvc** returns when the free disk space on the audit file system drops below this limit. Thus, the invoking program can take action to avoid running out of disk space. The **auditsvc** system call does not return until one of these 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.

A process must have **PRIV_SYS_AUDIT** in its set of effective privileges in order to execute this call successfully.

RETURN VALUES

auditsvc( ) returns only on an error.

ERRORS

EAGAIN	The descriptor referred to a *stream*, was marked for System V-style non-blocking I/O, and no data could be written immediately.
EBADF	*fd* is not a valid descriptor open for writing.
EBUSY	A second process attempted to perform this call.
ENOSPC	The user’s quota of disk blocks on the file system containing the file has been exhausted.
	Audit file-system space is below the specified limit.
EFBIG	An attempt was made to write a file that exceeds the file-size limit of the process or the maximum file size.
EINTR	The call is forced to terminate prematurely because of the arrival of a signal whose **SV_INTERRUPT** bit in **sv_flags** is set. [See **sigvec**(3B).] **signal (3C)** sets this bit for any signal it catches.
EINVAL	Auditing is disabled. [See **auditon**(2TSOL).]
	fd does not refer to a file of an appropriate type. Regular files are always appropriate.
EIO	An I/O error occurred while reading from or writing to the file system.

ENOSPC	There is no free space remaining on the file system containing the file.
ENXIO	A hangup occurred on the *stream* being written to.
EPERM	The process did not have the proper privilege in its effective set.
EWOULDBLOCK	The file was marked for 4.2BSD-style nonblocking I/O, and no data could be written immediately.

SUMMARY OF TRUSTED SOLARIS CHANGES

A process must have **PRIV_SYS_AUDIT** in its set of effective privileges in order to execute this call successfully.

SEE ALSO

auditd(1MTSOL), **audit**(2TSOL), **auditon**(2TSOL), **sigvec**(3B), **audit.log**(4TSOL)

NOTES

This functionality is active only if the audit module has been enabled. By default, this module is enabled on Trusted Solaris systems. See **bsmconv**(1MTSOL) for more information.

NAME

chdir, fchdir – Change working directory

SYNOPSIS

#include <unistd.h>

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

int fchdir(int *fildes***);**

MT-LEVEL

chdir( ) is Async-Signal-Safe.

DESCRIPTION

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

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

The information labels of *path* and of the calling process are unchanged.

RETURN VALUES

Upon successful completion, these functions return **0**. Upon failure, they return –**1** and set **errno** to indicate the error.

ERRORS

chdir( ) will fail and the current working directory will be unchanged if any of these conditions is true:

EACCES	Search permission is denied for some component of *path*. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_SEARCH** and **PRIV_FILE_MAC_SEARCH**.
EFAULT	*path* points to an illegal address.
EINTR	A signal was caught during the execution of the **chdir** function.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	Either a component of the path prefix or the directory named by *path* does not exist or is a null pathname.
ENOLINK	*path* points to a remote machine but the link to that machine is no longer active.
ENOTDIR	A component of the path name is not a directory.
EMULTIHOP	Components of *path* require hopping to multiple remote machines and file system type does not allow it.

fchdir( ) will fail and the current working directory will be unchanged if any of these conditions is true:

EACCES	Search permission is denied for *fildes*. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_SEARCH** and **PRIV_FILE_MAC_SEARCH**.
EBADF	*fildes* is not an open file descriptor.
EINTR	A signal was caught during the execution of the **fchdir( )** function.
EIO	An I/O error occurred while reading from or writing to the file system.
ENOLINK	*fildes* points to a remote machine but the link to that machine is no longer active.
ENOTDIR	The open file descriptor *fildes* does not refer to a directory.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

The information labels of *path* and of the calling process are unchanged.

SEE ALSO

chroot(2TSOL)

NAME

chmod, fchmod – Change access permission mode of file

SYNOPSIS

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

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

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

MT-LEVEL

chmod( ) is Async-Signal-Safe

DESCRIPTION

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

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

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

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

If the process is not a privileged process and the file is not a directory, mode bit 01000 (save text image on execution) is cleared. The calling process may assert the **PRIV_SYS_CONFIG** privilege to override this restriction.

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. The calling process may assert the **PRIV_SETID** privilege to override this restriction.

If a directory is writable and has **S_ISVTX** (the sticky bit) assigned, files within that directory can be removed or renamed only if any of these conditions is true [See **unlink**(2) and **rename**(2)C]:

- 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 assigned, 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 assigned and the mode bit 00010 (execute or search by group) is not assigned, mandatory file / record locking will exist on a regular file. This locking 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. The information label of *path* or *fildes* and of the calling process is unchanged.

RETURN VALUES

Upon success, the process returns a value of **0**. Upon failure, the process returns a value of −**1** and sets **errno** to indicate the error.

ERRORS

chmod( ) will fail and the file mode will be unchanged if any of these conditions prevails:

EACCES	Search permission is denied on a component of the path prefix of *path*. To override this restriction, the calling process may assert the **PRIV_FILE_DAC_SEARCH** privilege and / or the **PRIV_FILE_MAC_SEARCH** privilege.
	The calling process does not own the final object specified in *path* or does not own *fildes*. To override this restriction, the calling process may assert the **PRIV_FILE_SETDAC** privilege.
	Write permission is denied on *path* or *fildes*. To override this restriction, the calling process may assert the **PRIV_FILE_MAC_WRITE** privilege.
EFAULT	*path* points to an illegal address.
EINTR	A signal was caught during execution of the function.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines and file system type does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {

	_POSIX_NO_TRUNC} is in effect.
ENOENT	A component of the path prefix or the file referred to by *path* either does not exist or is a null pathname.
ENOLINK	*fildes* points to a remote machine but 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 the calling process is not privileged. To override this restriction, the calling process may assert the **PRIV_FILE_SETDAC** privilege.
EROFS	The file referred to by *path* resides on a read-only file system.

fchmod( ) will fail and the file mode will be unchanged if if any of these conditions prevails:

EBADF	*fildes* is not an open file descriptor.
EIO	An I/O error occurred while reading from or writing to the file system.
EINTR	A signal was caught during execution of the **fchmod**( ) function.
ENOLINK	*path* points to a remote machine but 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.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

To set the set-group-ID bit on a group not in effective or supplementary group IDs of the calling process, the calling process may assert the **PRIV_SYS_CONFIG** privilege.

To set the sticky bit on a file, the calling process may assert the **PRIV_SETID** privilege.

The information label of *path* or *fildes* and of the calling process is unchanged.

SEE ALSO

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

System Interface Guide

NAME

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

SYNOPSIS

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

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

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

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

MT-LEVEL

chown( ) is Async-Signal-Safe.

DESCRIPTION

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

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

If **chown**, **lchown**, or **fchown( )** is invoked, the set-user-ID and set-group-ID bits of the file mode, **S_ISUID** and **S_ISGID** respectively, are cleared. [See **chmod**(2TSOL).] To bypass this restriction, the process may assert the PRIV_FILE_SETID privilege.

The operating system has a configuration option, {**_POSIX_CHOWN_RESTRICTED**}, to restrict ownership changes for the **chown**, **lchown**, and **fchown** functions. When {**_POSIX_CHOWN_RESTRICTED**} is not in effect, the effective user ID of the process must match the owner of the file. To override this restriction, the calling process must assert the PRIV_FILE_CHOWN privilege. When {**_POSIX_CHOWN_RESTRICTED**} is in effect, the **chown**, **lchown**, and **fchown** functions require that the calling process assert the PRIV_FILE_CHOWN privilege to change the user ID of a file. To change the group ID of a file, the process must be the owner of the file and the new group ID must be the group of the process ID or must be in the supplementary group list of the process. To override this restriction, the calling process may assert the PRIV_FILE_CHOWN privilege.

Upon successful completion, **chown**, **fchown**, and **lchown** mark for update the **st_ctime** field of the file. The information label of *path* or *fildes* and the information label of the calling process remain unchanged.

RETURN VALUES

Upon successful completion, these functions return a value of **0**. Upon failure, they return a value of –**1** and set **errno** to indicate the error.

ERRORS

chown and **lchown** fail and the owner and group of the named file remain unchanged if any of these conditions is true:

EACCES	Search permission is denied on a component of the path prefix of *path*. To override this restriction, the calling process may assert one or both of these privileges: PRIV_FILE_DAC_SEARCH and PRIV_FILE_MAC_SEARCH.

	Write permission is denied on *path* or *fildes*. To override this restriction, the calling process may assert the PRIV_FILE_MAC_WRITE privilege.
EFAULT	*path* points to an illegal address.
EINTR	A signal was caught during the **chown** or **lchown** functions.
EINVAL	*group* or *owner* is out of range.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines but the file system type does not allow it. Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOLINK	*path* points to a remote machine but the link to that machine is no longer active.
ENOENT	Either a component of the path prefix or the file to which *path* refers does not exist or is a null pathname.
ENOTDIR	A component of the path prefix of *path* is not a directory.
EPERM	The effective user ID does not match the owner of the file. If {**_POSIX_CHOWN_RESTRICTED**} is set, the calling process must assert the PRIV_FILE_CHOWN privilege. If {**_POSIX_CHOWN_RESTRICTED**} is not set, the calling process may assert the PRIV_FILE_CHOWN privilege.
EROFS	The named file resides on a read-only file system.

fchown fails and the owner and group of the named file remain unchanged if any of these conditions is true:

EACCES	*fildes* is opened for read, but mandatory write access is required to perform this operation.
EBADF	*fildes* is not an open file descriptor.
EIO	An I/O error occurred while reading from or writing to the file system.
EINTR	A signal was caught during execution of the function.
ENOLINK	*fildes* points to a remote machine but the link to that machine is no longer active.
EINVAL	*group* or *owner* is out of range.
EPERM	The effective user ID does not match the owner of the file or the process is not privileged and {**_POSIX_CHOWN_RESTRICTED**}

indicates that privilege is required. To override this restriction, the calling process may assert the PRIV_FILE_CHOWN privilege.

EROFS The named file to which *fildes* refers resides on a read-only file system.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

When the ownership of *path* and *fildes* is changed, the set-user-ID and set-group-ID bits are cleared. The calling process may assert the PRIV_FILE_SETID privilege to bypass this restriction.

To change the user ID of the file when the calling process does not own the file and {**_POSIX_CHOWN_RESTRICTED**} is not in effect, the calling process may assert the PRIV_FILE_CHOWN privilege.

To change the group ID of the file when the calling process does not own the file, and the new group ID is not in the group ID of the process or in the supplementary group list of the process, and {**_POSIX_CHOWN_RESTRICTED**} is not in effect, the calling process may assert the PRIV_FILE_CHOWN privilege.

The information label of *path* or *fildes* and the information label of the calling process remain unchanged.

SEE ALSO

chgrp(1), **chown**(1), **chmod**(2TSOL)

NAME

chroot, fchroot – Change root directory

SYNOPSIS

#include <unistd.h>

int chroot(const char **path***);**

int fchroot(int *fildes***);**

DESCRIPTION

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

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

The calling process must assert the **PRIV_PROC_CHROOT** privilege to use this system call. **fchroot** is further restricted: although changing to the system root using this call is always possible, change is not guaranteed to succeed in any other case, even when *fildes* is valid in all respects.

The ‘‘**..**’’ entry in the root directory is interpreted as the root directory itself. Thus, ‘‘**..**’’ cannot be used to access files outside the subtree rooted at the root directory. Instead, **fchroot** can be used to set the root back to a directory that was opened before the root directory was changed.

The information labels of *path* and of the calling process are unchanged.

RETURN VALUES

Upon successful completion, these functions return **0**. Upon failure, they return –**1** and set **errno** to indicate the error.

ERRORS

chroot will fail and the root directory will remain unchanged if any of these conditions is true:

EACCES	Search permission is denied for a component of the path prefix of *path*. Search permission is denied for the directory referred to by *path*. To override these restrictions, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_SEARCH** and **PRIV_FILE_MAC_SEARCH**.
EBADF	The descriptor is not valid.
EFAULT	*path* points to an illegal address.
EINVAL	**fchroot** attempted to change to a directory that is not the system root and external circumstances do not allow this.
EINTR	A signal was caught during the **chroot** function.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.

EMULTIHOP	Components of *path* require hopping to multiple remote machines and file system type does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	The named directory does not exist or is a null pathname.
ENOLINK	*path* points to a remote machine but the link to that machine is no longer active.
ENOTDIR	A component of the path name is not a directory.
EPERM	The calling process must assert the **PRIV_PROC_CHROOT** privilege to change the root directory.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

The calling process must assert the **PRIV_PROC_CHROOT** privilege to change the root directory.

The information labels of *path* and of the calling process are unchanged.

SEE ALSO

chroot(1MTSOL)

WARNINGS

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

NAME

chstate – Change the view of a host state between labeled and unlabeled

SYNOPSIS

#include <tsol/tndb.h>
int chstate(tsol_chstateop_t *state***, struct netbuf** ∗*addr***);**

DESCRIPTION

A host regards another host as labeled or unlabeled, based on the remote host's database caches that are loaded in the kernel. In some cases (for example, when a diskless client boots), the server host must initially regard the client as an unlabeled host even though the client is a labeled host; at a later time, the server host can regard the client as a labeled host. **chstate()** allows a process to toggle the view of a host between labeled and unlabeled.

The argument *state* is of the following type:

```
typedef enum {
        STATE_UNLABELED = 1,
        STATE_LABELED   = 2
} tsol_chstateop_t;
```

The argument *addr* is a pointer to the **netbuf** structure:

```
struct netbuf {
        unsigned int maxlen;
        unsigned int len;
        char ∗buf;
};
```

where *buf* contains the address of the host whose view is being changed. Currently, only the IP address format is supported; and it should be specified as type **sockaddr_in**.

chstate requires the PRIV_SYS_NET_CONFIG privilege.

RETURN VALUES

Upon successful completion, the process returns a value of **0**. Otherwise, the process returns a value of –**1** and sets **errno** to indicate the error.

ERRORS

chstate( ) may fail for one of these reasons:

EFAULT	The *addr* argument points to a bad address.
EINVAL	Either the *state* argument is not one of the listed type constants, or the remote host template for the host specified by *addr* is not available (after using fall-back mechanism).
EPERM	The calling process does not have the PRIV_SYS_NET_CONFIG privilege.

NAME

creat – Create a new file or rewrite an existing one

SYNOPSIS

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

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

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

creat( ) creates a new ordinary file or prepares to rewrite an existing file to which *path* points.

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. The calling process may assert the **PRIV_FILE_SETID** privilege to override clearing of the S_ISGID bit.
- All bits set in the file mode-creation mask of the process are cleared. [See **umask**(2).]
- The "save text image after execution bit" of the mode is cleared. [See **chmod**(2TSOL) for the values of *mode*.] The calling process may assert the **PRIV_SYS_CONFIG** privilege to override the clearing of the **S_ISVTX** bit.

If the file exists, its sensitivity label is unchanged. If the file does not exist, it is created with its sensitivity label set to the sensitivity label of the calling process.

The information label of *path* is set to ADMIN_LOW.

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, the function returns a nonnegative integer, the lowest-numbered, unused file descriptor. Upon failure, the function returns –**1**, no files are created or modified, and **errno** is set to indicate the error.

ERRORS

creat fails if any of these conditions is true:

EACCES	Search permission is denied on a component of the path prefix. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_SEARCH** and **PRIV_FILE_MAC_SEARCH**.
	The file does not exist and the directory in which the file is to be created does not permit writing. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_WRITE** and **PRIV_FILE_MAC_WRITE**.
	The file exists and write permission to *path* is denied. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_MAC_WRITE** and **PRIV_FILE_DAC_WRITE**.
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 in which 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 user's quota of inodes on the file system in which the file is being created has been exhausted.
EFAULT	*path* points to an illegal address.
EINTR	A signal was caught during the **creat** function.
EISDIR	The named file is an existing directory.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMFILE	The process has too many open files. [See **getrlimit**(2TSOL).]
EMULTIHOP	Components of *path* require hopping to multiple remote machines.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENFILE	The system file table is full.
ENOENT	A component of the path prefix does not exist.
	The path name is null.
ENOLINK	*path* points to a remote machine but the link to that machine is no longer active.
ENOSPC	The file system is out of inodes.
ENOTDIR	A component of the path prefix is not a directory.
EROFS	The named file resides or would reside on a read-only file system.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

To override clearing of the S_ISVTX bit, the calling process may assert the **PRIV_SYS_CONFIG** privilege. To override the clearing of the S_ISGID bit, the calling process may assert the **PRIV_FILE_SETID** privilege.

If *path* exists, its sensitivity label is unchanged. If *path* does not exist, it is created with its sensitivity label set to the sensitivity label of the calling process.

The information label of *path* is set to ADMIN_LOW and does not float the information label of the containing directory.

SEE ALSO

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

NAME

devpolicy – get / set device driver policy table

SYNOPSIS

cc [flag ...] file
#include <sys/tsol/devpolicy.h>
int devpolicy(devpolicy_op_t *op***, devpolicy_t** **tbl* **, int** **len***)**

DESCRIPTION

devpolicy sets and gets the device policy table.

Allowed values for *op* are specified in **<sys/tsol/devpolicy.h>** and may be one of the following:

TSOL_GET_DEVPOLICY	Get the device policy table. The *tbl* argument points to a buffer containing the **devpolicy_t** array, and *len* contains the length of the array. **devpolicy** returns in *len* the number of elements that the kernel has filled in the array.
TSOL_SET_DEVPOLICY	Set the device policy table. The *tbl* argument points to the **devpolicy_t** structure to be downloaded to the kernel, and *len* contains the length of the array. For this call to succeed, the calling process must have **PRIV_SYS_DEVICES** in its set of effective privileges.

SEE ALSO

devpolicy(1MTSOL)

NAME

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

SYNOPSIS

#include <unistd.h>

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

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

int execle (const char **path*,**char *const** *arg0*[ ]**, . . . , const char** **argn*,
 char * /*NULL*/, **char *const** *envp*[ ]**);**

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

int execlp (const char **file*, **const char** **arg0*, **. . ., const char** **argn*, **char** * /*NULL*/**);**

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

DESCRIPTION

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

An interpreter file begins with a line of the form

#! *pathname* [*arg*]

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

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

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

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

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

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

Solaris	**exec** uses **/usr/bin/sh** [See **sh**(1).]
XPG4	**exec** uses the XPG4-compliant shell **/usr/bin/ksh** [See **ksh**(1).]

The calling process must have read and execute access to the new process file or have the following in its set of effective privileges:

PRIV_FILE_DAC_SEARCH
PRIV_FILE_DAC_EXECUTE
PRIV_FILE_MAC_SEARCH
PRIV_FILE_MAC_READ

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

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

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

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

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

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

If the process has the **PRIV_PROC_OWNER** privilege, the set-user-ID and set-group-ID bits will be honored when the process is being controlled by **ptrace**.

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

Profiling is disabled for the new process. [See **profil**(2).]

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

Any outstanding asynchronous I/O operations may be cancelled.

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

nice value [See **nice**(2).]
scheduler class and priority [See **priocntl**(2).]
process ID
parent process ID
process group ID
supplementary group IDs
semadj values [See **semop**(2TSOL).]
session ID [See **exit**(2) and **signal**(3C).]
trace flag [See **ptrace**(2) request 0)
time left until an alarm [See **alarm**(2).]
current working directory
root directory
file mode creation mask [See **umask**(2).]
resource limits [See **getrlimit**(2TSOL).]
utime, **stime**, **cutime**, and **cstime** [See **times**(2).]
file-locks [See **fcntl**(2TSOL) and **lockf**(3C).]
controlling terminal
process signal mask [See **sigprocmask**(2).]
pending signals [See **sigpending**(2).]
clearance [See **getclearance**(2TSOL)C]
sensitivity label [See **getcmwlabel**(2TSOL)C]
inheritable privilege set [See **getppriv**(2TSOL)C]
process attribute flags [See **getpattr**(2TSOL)C]

The process information label of the new process is either inherited from the calling process or reset to ADMIN_LOW depending on the system configuration. In either case, the process information label may be floated by the program file when it is loaded for execution.

The four privilege sets of the new process are updated as described in the following equations where E1, P1, S1, I1 are the four privilege sets of the calling process; E2, P2, S2, I2 are the four privilege sets of the new process; and F and A are the forced set and the allowed set of the program file:

E2 = P2 = (I1 union F) intersect A
S2 = I1 intersect A
I2 = I1

When a script file is run, the resulting forced privileges are the combination of the forced privileges of the script and the forced privileges of the interpreter program; and the resulting allowed privileges are the allowed privileges of the interpreter program. The privilege update equations for a script executable could be expressed like this:

E2 = P2 = (I1 union Fs union Fi) intersect Ai
S2 = I1 intersect A
I2 = I1

where

Fs is the forced privilege set of the script,
Fi is the forced privilege set of the interpreter program, and
Ai is the allowed privilege set of the interpreter program.

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

RETURN VALUES

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

ERRORS

exec will fail and return to the calling process if any of these conditions is true:

E2BIG	The number of bytes in the new process's argument list is greater than the system-imposed limit of 5120 bytes. The argument list limit is the 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 path prefix of the new process file. Moreover, the calling process does not have **PRIV_FILE_DAC_SEARCH** and / or **PRIV_FILE_MAC_SEARCH** to override the restriction.
	The new process file is not an ordinary file.
	The new process file mode denies execute permission.
EAGAIN	Total amount of system memory available when reading using raw I/O is temporarily insufficient.
EFAULT	An argument points to an illegal address.
EINTR	A signal was caught during the **exec** function.
ELOOP	Too many symbolic links were encountered in translating *path* or *file.*
EMULTIHOP	Components of *path* require hopping to multiple remote machines and the file system type does not allow it.
ENAMETOOLONG	The length of the *file* or *path* argument exceeds {**PATH_MAX**}, or the length of a *file* or *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	One or more components of the new process path name of the file do not exist or the path name is null.
ENOEXEC	The **exec** is not an **execlp**( ) or **execvp**( ), and the new process file has the appropriate access permission but an invalid magic number in its header.
ENOLINK	*path* points to a remote machine but the link to that machine is no

longer active.

ENOMEM	The new process requires more memory than **RLIMIT_VMEM**, the limit imposed by **setrlimit**(). [See **brk**(2).]
ENOTDIR	A component of the new process path of the file prefix is not a directory.

SUMMARY OF TRUSTED SOLARIS CHANGES

MAC search and execute permissions on the executable object are required. Process privilege sets and process information label are updated upon execution of the program. Other Trusted Solaris process attributes, such as clearance, sensitivity label, and process attribute flags, are unchanged.

SEE ALSO

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

NAME

fcntl – Control files

SYNOPSIS

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

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

MT-LEVEL

Async_Signal_Safe

DESCRIPTION

fcntl( ) provides for control over open files. *fildes* is an open file descriptor. [See **intro**(2TSOL).]

fcntl may take a third argument, *arg*, whose data type, value, and use depend upon the value of *cmd*. *cmd* specifies one of these operations to be performed by **fcntl**.

F_DUPFD Return a new file descriptor with these characteristics:

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

Same open file (or pipe) as the original file

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

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

Shares any locks associated with the original file descriptor

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

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

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

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

F_GETFL Get *fildes* status flags.

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

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

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

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

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

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

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

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

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

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

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

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

```
typedef struct flock {
        short     l_type;
        short     l_whence;
        off_t     l_start;
        off_t     l_len;       /* len == 0 means until end of file */
        long      l_sysid;
        pid_t     l_pid;
        long        pad[4];      /* reserve area */
} flock_t;
```

The **flock** structure describes the type (**l_type**), starting offset (**l_whence**), relative offset (**l_start**), size (**l_len**), process ID (**l_pid**), and system ID (**l_sysid**) of the segment of the file to be affected. The process ID and system ID fields are used only with the **F_GETLK**

NAME

fork, fork1 – Create a new process

SYNOPSIS

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

pid_t fork(void);

pid_t fork1(void);

MT-LEVEL

fork( ) is Async-Signal-Safe.

DESCRIPTION

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

- real user ID, real group ID, effective user ID, effective group ID
- environment
- open file descriptors
- close-on-exec flags [See **exec**(2TSOL).]
- 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**(2TSOL).]
- scheduler class [See **priocntl**(2TSOL).]
- all attached shared memory segments [See **shmop**(2TSOL).]
- 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**(2TSOL).]
- controlling terminal
- saved user ID and group ID
- effective privilege set [See **intro**(2TSOL).]
- permitted privilege set [See **intro**(2TSOL).]
- inheritable privilege set [See **intro**(2TSOL)]
- saved privilege set [See **intro**(2TSOL).]
- process attribute flags [See **getpattr**(2TSOL).]
- clearance [See **intro**(2TSOL).]
- sensitivity label [See **intro**(2TSOL).]
- information label [See **intro**(2TSOL).]

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**(2TSOL).] The child process differs from the parent process in the following ways:

- The child process has a unique process ID that does not match any active process group ID.
- The child process has a different parent process ID (that is, the process ID of the parent process).
- The child process has its own copy of the parent's file descriptors and directory streams. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.
- Each shared memory segment remains attached and the value of **shm_nattach** is incremented by 1.
- All **semadj** values are cleared. [See **semop**(2TSOL).]
- Process locks, text locks, data locks, and other memory locks are not inherited by the child. [See **plock**(3C) and **memcntl**(2).]
- The child process's **tms** structure is cleared: **tms_utime**, **stime**, **cutime**, and **cstime** are set to 0. [See **times**(2).]
- The child processes resource utilizations are set to 0. [See **getrlimit**(2TSOL).] The **it_value** and **it_interval** values for the **ITIMER_REAL** timer are reset to 0. [See **getitimer**(2).]
- The set of signals pending for the child process is initialized to the empty set.
- Timers created by **timer_create**(3R) are not inherited by the child process.
- No asynchronous input nor asynchronous output operations are inherited by the child.

Record locks set by the parent process are not inherited by the child process. [See **fcntl**(2TSOL).]

MT fork( ) Solaris Threads

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

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

POSIX Threads

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

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

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

Fork-safety

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

RETURN VALUES

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

ERRORS

fork fails and no child process is created if any of these conditions is true:

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

The system-imposed limit on the total number of processes under execution by a single user would be exceeded, and the calling process does not have the **PRIV_SYS_MAXPROC** effective privilege.

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

ENOMEM There is not enough swap space.

SUMMARY OF TRUSTED SOLARIS CHANGES

Process attributes introduced by Trusted Solaris are all inheritable by the child process. A calling process with the **PRIV_SYS_MAXPROC** privilege is able to override the limit on the number of processes a user may have.

SEE ALSO

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

NOTES

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

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

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

NAME

fpathconf, pathconf – Get configurable path-name variables

SYNOPSIS

#include <unistd.h>

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

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

MT-LEVEL

pathconf( ) is Async-Signal-Safe.

DESCRIPTION

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

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

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

NOTES:

1) If *path* or *fildes* refers to a directory, the value returned applies to the directory itself.
2) The behavior is undefined if *path* or *fildes* does not refer to a terminal file.
3) If *path* or *fildes* refers to a directory, the value returned applies to the file names within the directory.
4) The behavior is undefined if *path* or *fildes* does not refer to a directory.

5) If *path* or *fildes* refers to a directory, the value returned is the maximum length of a relative path name when the specified directory is the working directory.

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

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

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

The information label of *path* or *fildes* and the information label of the calling process remain unchanged.

RETURN VALUES

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

ERRORS

fpathconf fails if either of these conditions is true:

EBADF	*fildes* is not a valid file descriptor.
EACCES	*fildes* is open only for writing and the calling process does not have mandatory read access to the object to which the descriptor refers. To override this restriction, the calling process may assert the **PRIV_FILE_MAC_READ** privilege.

pathconf fails if any of these conditions is true:

EACCES	Search permission is denied for a component of the path prefix. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_SEARCH** and **PRIV_FILE_MAC_SEARCH**.
	The calling process does not have mandatory read access to *path*. To override this restriction, the calling process may assert the **PRIV_FILE_MAC_READ** privilege.
ELOOP	Too many symbolic links are encountered while translating *path*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines but the file system type does not allow it.
ENAMETOOLONG	The length of a path name exceeds **{PATH_MAX}**, or a path name component is longer than **{NAME_MAX}** while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	*path* is needed for the command specified; and the named file does

not exist or the *path* argument points to an empty string.

ENOLINK	*path* points to a remote machine but the link to that machine is no longer active.
ENOTDIR	A component of the path prefix is not a directory.

Both **fpathconf** and **pathconf** fail if this condition is true:

EINVAL	*name* is an invalid value.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

The information label of *path* or *fildes* and the information label of the calling process remain unchanged.

SEE ALSO

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

NAME

getaudit, setaudit – Get and set process audit information

SYNOPSIS

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

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

int getaudit(struct auditinfo **info***);**

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

DESCRIPTION

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

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

The **auditinfo** structure used to pass the process audit information contains these 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 */
```

To execute these commands successfully, a process needs certain privileges in its set of effective privileges: for **getaudit**, a process needs **PRIV_SYS_AUDIT**, **PRIV_PROC_AUDIT_TCB**, or **PRIV_PROC_AUDIT_APPL**; for **setaudit**, **PRIV_SYS_AUDIT**.

RETURN VALUES

Upon success, **getaudit** and **setaudit** return **0**. Upon failure, they return –**1** and set **errno** to indicate the error.

ERRORS

EFAULT	The *info* parameter points outside the allocated address space of the process.
EPERM	The process did not have the appropriate privilege.

SUMMARY OF TRUSTED SOLARIS CHANGES

As explained in DESCRIPTION, privileges are needed to run this command successfully.

SEE ALSO

audit(2TSOL)

NOTES

This functionality is active only if the audit module has been enabled. By default, this module is enabled on Trusted Solaris systems. See **bsmconv**(1MTSOL) for more information.

Trusted Solaris 2.x will soon extend the number of audit classes and introduce new but similar structures and programming interfaces.

NAME

getauid, setauid – Get and set user audit identity

SYNOPSIS

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

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

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

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

DESCRIPTION

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

The **setauid** system call sets the audit user ID for the current process.

Only a process with the **PRIV_SYS_AUDIT** privilege asserted may successfully set its user identity. To get its identity successfully, a process must have **PRIV_SYS_AUDIT**, **PRIV_PROC_AUDIT_TCB**, or **PRIV_PROC_AUDIT_APPL** in its set of effective privileges.

RETURN VALUES

Upon success, **getauid** returns the audit user ID of the current process. Upon failure, **getauid** returns **–1** and sets **errno** to indicate the error.

Upon success, **setauid** returns **0**. Upon failure, **setauid** returns **–1** and sets **errno** to indicate the error.

ERRORS

EFAULT	*auid* points to an invalid address.
EPERM	The process lacks the privilege necessary to use the command.

SUMMARY OF TRUSTED SOLARIS CHANGES

The privileges explained in **DESCRIPTION** are needed to run this command successfullly.

SEE ALSO

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

NOTES

This functionality is active only if the audit module has been enabled. By default, this module is enabled on Trusted Solaris systems. See **bsmconv**(1MTSOL) for more information.

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

NAME

getclearance – Get process clearance

SYNOPSIS

cc [*flag* ...] *file* ... **–ltsol** [*library* ...]

#include <tsol/label.h>

int getclearance(bclear_t **clearance_p***)**

DESCRIPTION

getclearance( ) obtains the clearance of the calling process. The clearance information is placed into the memory pointed to by *clearance_p*.

RETURN VALUES

Upon success, **getclearance** returns **0**. Upon failure, **getclearance** returns –**1** and sets **errno** to indicate the error.

ERRORS

getclearance will fail (and *clearance_p* will not refer to a valid clearance) if this condition is true:

EFAULT	*clearance_p* points to an invalid address.

SEE ALSO

setclearance(2TSOL)

NAME

getcmwfsrange, fgetcmwfsrange – get file system sensitivity label range

SYNOPSIS

#include <tsol/label.h>

int getcmwfsrange(char **path*, **brange_t** **range_p***)**

int fgetcmwfsrange(int *fd*, **brange_t** **range_p***)**

DESCRIPTION

getcmwfsrange( ) returns the sensitivity label range of a mounted file system. *path* is the path name of any file within the mounted filesystem. *range_p* is a pointer to a sensitivity label range structure defined as follows:

```
struct binary_level_range {
        blevel_t lower_bound;
        blevel_t upper_bound;
};
typedef struct binary_level_range brange_t;     /* Level Range */
```

fgetcmwfsrange( ) returns the same information about an open file referred to by descriptor *fd*.

The information label of *path* or *fd* is unchanged. The information label of the calling process is also unchanged.

RETURN VALUES

getcmwfsrange( ) and **fgetcmwfsrange( )** return:

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

ERRORS

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

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

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

EBADF	*fd* is not a valid open file descriptor.
EFAULT	*range_p* points to an invalid address.
EINVAL	*fd* refers to a socket, not a file.
EIO	An I/O error occurred while reading from the file system.

NAME

getcmwlabel, lgetcmwlabel, fgetcmwlabel – get file CMW label

SYNOPSIS

#include <tsol/label.h>

int getcmwlabel(char **path*, **bclabel_t** **label_p*

int fgetcmwlabel(int *fd*, **bclabel_t** **label_p***)**

DESCRIPTION

getcmwlabel( ) obtains the CMW label of the file named by *path*. Mandatory read access to the final component of *path* is required or the calling process must have PRIV_FILE_MAC_READ in its set of effective privileges. Discretionary read, write or execute permission to the final component of *path* is not required, but all directories in the path prefix of *path* must be searchable.

If *path* refers to a FIFO, then the CMW label associated with the FIFO is returned. The information label portion of *label_p* returned by this interface does not vary with the information label associated with any data that is present in the FIFO.

If *path* refers to a directory, then the information label portion is undefined.

lgetcmwlabel( ) is like **getcmwlabel( )** except in the case where the final component of *path* is a symbolic link, in which case **lgetcmwlabel( )** returns the CMW label of the link, while **getcmwlabel( )** returns the CMW label of the file to which the link refers.

fgetcmwlabel( ) obtains the CMW label of an open file referred to by the argument descriptor, such as would be obtained by an **open**(2V) call. If the descriptor is only open for writing, then mandatory read access to the object is required or the calling process must have PRIV_FILE_MAC_READ in its set of effective privileges.

If the descriptor refers to a FIFO, then the information label portion of *label_p* refers to the information label of the last data that was read. If no data has been read, then the information label portion is set to the ADMIN_LOW information label.

label_p is a pointer to a CMW label structure defined as follows:

```
struct binary_cmw_label {
        bslabel_t   bcl_sensitivity_label;
        bilabel_t   bcl_information_label;
} ;

typedef struct binary_cmw_label bclabel_t;/* CMW Label */
```

An exception to the access rules applies in the case of **pty** pseudo-terminals (**/dev/ptyp*** and **/dev/ttyp***). Normally mandatory read access is required or the calling process must have PRIV_FILE_MAC_READ in its set of effective privileges. If the specified file is a **pty** device file and the calling process does not have mandatory read access or PRIV_FILE_MAC_READ is not in its set of effective privileges, each function returns success and sets *label_p* to the ADMIN LOW sensitivity label and the ADMIN LOW information label.

RETURN VALUES

getcmwlabel(), **lgetcmwlabel()** and **fgetcmwlabel()** return:

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

ERRORS

getcmwlabel() and **lgetcmwlabel()** fail if one or more of the following are true:

EACCES Search permission is denied for a component of the path prefix of *path*. To override this restriction, the calling process may assert the PRIV_FILE_DAC_SEARCH privilege and / or the PRIV_FILE_MAC_SEARCH privilege.

The calling process does not have mandatory read access to *path* because the sensitivity label of the calling process does not dominate the sensitivity label of the final component of *path* and the calling process does not have PRIV_FILE_MAC_READ in its set of effective privileges.

EFAULT *label_p* or *path* points to an invalid address.

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

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

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

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

ENOENT The file referred to by *path* does not exist.

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

EPERM The calling process does not have mandatory read access to *path* because the sensitivity label of *path* is outside the calling processes' clearance and the calling process does not have PRIV_FILE_MAC_READ in its set of effective privileges.

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

EACCES The descriptor is only open for writing and the calling process does not have mandatory read access to the object referred to by the descriptor because the sensitivity label of the calling process does not dominate the sensitivity label of the object and the calling process does not have PRIV_FILE_MAC_READ in its set of effective privileges.

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

EFAULT *label_p* points to an invalid address.

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

SEE ALSO

open(2V), **setcmwlabel**(2TSOL)

NAME

getcmwplabel – Get process CMW label

SYNOPSIS

cc [*flag* ...] *file* ... **–ltsol** [*library* ...]

#include <tsol/label.h>

int getcmwplabel(bclabel_t **label_p***)**

DESCRIPTION

getcmwplabel( ) obtains the CMW label of the calling process. The label information is placed into the memory to which *label_p* points.

RETURN VALUES

Upon success, **getcmwplabel** returns **0**. Upon failure, **getcmwplabel** returns –**1** and sets **errno** to indicate the error.

ERRORS

getcmwplabel fails (and *label_p* does not refer to a valid CMW label) if this condition is true:

EFAULT	*label_p* points to an invalid address.

SEE ALSO

setcmwplabel(2TSOL)

NAME

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

SYNOPSIS

#include <sys/dirent.h>

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

DESCRIPTION

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

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

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

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

The information label of the object is unchanged. If information label floating is turned on for the system, the information label of the calling process floats according to the information label of *fildes*. To stop the floating, the calling process may assert the **PRIV_PROC_NOFLOAT** privilege.

RETURN VALUES

Upon successful completion, **getdents** returns a nonnegative integer indicating the number of bytes actually read. A value of **0** indicates that the end of the directory has been reached. If the function fails, it returns –**1** and sets **errno** to indicate the error.

ERRORS

getdents will fail if any of these conditions is true:

EACCESS	The calling process is not allowed to read the *procfs* file system. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_READ** and **PRIV_FILE_MAC_READ**.
	The system is configured to check mandatory read access to the directory entries. To override this restriction, the calling process may assert the **PRIV_FILE_MAC_READ** privilege.
EBADF	*fildes* is not a valid file descriptor open for reading.
EFAULT	*buf* points to an illegal address.
EINVAL	*nbyte* is not large enough for one directory entry.
EIO	An I/O error occurred while accessing the file system.
ENOENT	The current file pointer for the directory is not located at a valid entry.
ENOLINK	*fildes* points to a remote machine but the link to that machine is no longer active.
ENOTDIR	*fildes* is not a directory.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

To prevent the process information label from floating, the calling process may assert the **PRIV_PROC_NOFLOAT** privilege.

SEE ALSO

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

NAME

fgetfattrflag, fsetfattrflag, getfattrflag, setfattrflag, mldgetfattrflag, mldsetfattrflag – set / get the security attribute flags of a file

SYNOPSIS

#include <tsol/secflgs.h>

int getfattrflag(const char **path*, **secflgs_t** **flags*);

int setfattrflag(const char **path*, **secflgs_t** *which*, **secflgs_t** *flags*);

int fgetfattrflag(int *fildes*, **secflgs_t** **flags*);

int fsetfattrflag(int *fildes*, **secflgs_t** *which*, **secflgs_t** *flags*);

int mldgetfattrflag(const char **path*, **secflgs_t** **flags*);

int mldsetfattrflag(const char **path*, **secflgs_t** *which*, **secflgs_t** *flags*);

DESCRIPTION

setfattrflag( ) , **fsetfattrflag( )** , and **mldsetfattrflag( )** set the security flags of the file whose name is given by *path* or referred to by the open file descriptor *fildes* . The bit pattern contained in *which* is used to indicate which flags are being affected. The corresponding bits in *flags* are set to 1 or 0 to indicate whether the affected flags are being set or unset respectively.

getfattrflag( ) , **fgetfattrflag( )** , and **mldgetfattrflag( )** get the security flags of the file whose name is given by *path* or referred to by the open file descriptor *fildes* and store it in the location pointed to by *flags*.

Attribute bits are interpreted as follows:

FAF_MLD	Directory has MLD semantics.
FAF_PUBLIC	Filesystem object is a public object.
FAF_SLD	Directory is an SLD.

Attribute flags are constructed by **OR** 'ing the attribute flag bits.

FAF_MLD is the only flag that may be modified without privilege if the directory is empty, the effective user ID of the process matches the directory owner, and the process has mandatory as well as discretionary write access. The **FAF_MLD** flag, once set, cannot be unset. Additionally, the **FAF_MLD** flag may only be set via the **mldsetfattrflag()** interface. The **FAF_PUBLIC** flag can only be read or modified by a process possessing the PRIV_FILE_AUDIT privilege. A process attempting to read the **FAF_PUBLIC** flag without the PRIV_FILE_AUDIT privilege in effect will not fail. However the value of **FAF_PUBLIC** will be returned as unset. The **FAF_SLD** flag can never be set. The ability to read any flag is dependant upon the process having mandatory and discretionary read access to the file. The ability to set any flag is dependant upon the process having mandatory and discretionary write access to the file.

If *path* is a symbolic link, the target's attribute flags are affected rather than the link's. If *path* is a multilevel directory, **getfattrflag()** and **setfattrflag()** will affect the underlying single-level directory beneath (unless *path* is adorned). **mldgetfattrflag()** and **mldsetfattrflag()** do not translate multi-level directories to underlying single-level directories. **fgetfattrflag()** and **fsetsattrflag()** affect only the file referred to by *fildes* .

The information label of *path* or *fildes* is unchanged. The information label of the calling process is also unchanged.

RETURN VALUES

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

ERRORS

getfattrflag( ) and **mldgetfattrflag( )** will fail if one or more of the following are true:

EACCES	Search permission is denied on a component of the path prefix of *path*. To override this restriction, the calling process may assert the PRIV_FILE_DAC_SEARCH privilege and / or the PRIV_FILE_MAC_SEARCH privilege.
EACCES	Read permission is denied the final component of *path*. To override this restriction, the calling process may assert the PRIV_FILE_MAC_READ privilege.
EFAULT	*path* points to an illegal address.
EINTR	A signal was caught during execution of the function.
EIO	An I/O error occurred while reading from the file system.
ELOOP	Too many symbolic links were encountered in translating *path/c* .
EMULTIHOP	Components of *path* require hopping to multiple remote machines and file system type does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	Either a component of the path prefix, or the file referred to by *path* does not exist or is a null pathname.
ENOLINK	*fildes* points to a remote machine and the link to that machine is no longer active.
ENOTDIR	A component of the prefix of *path* is not a directory.

fgetfattrflag( ) fails and the file mode is unchanged if:

EACCES	Read permission is denied on *fildes.* To override this restriction, the calling process may assert the PRIV_FILE_MAC_READ privilege.
EBADF	*fildes* is not an open file descriptor
EIO	An I/O error occurred while reading from the file system.
EINTR	A signal was caught during execution of the **fgetfattrflag( )** function.

setfattrflag( ) and **mldsetfattrflag( )** will fail and the file mode is unchanged if one or more of the following are true:

EACCES	Search permission is denied on a component of the path prefix of *path*. To override this restriction, the calling process may assert

	the PRIV_FILE_DAC_SEARCH privilege and / or the PRIV_FILE_MAC_SEARCH privilege.
EACCES	Write permission is denied *path*. To override this restriction, the calling process may assert the PRIV_FILE_MAC_WRITE privilege.
EACCES	The calling process does not own *fildes*. To override this restriction, the calling process may assert the PRIV_FILE_OWNER privilege.
EFAULT	*path* points to an illegal address.
EINTR	A signal was caught during execution of the function.
EINVAL	*path* is not a valid pathname. When setting **FAF_MLD**, *path* must refer to an empty directory.
EIO	An I / O error occurred while writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines and file system type does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds **{ PATH_MAX },** or the length of a *path* component exceeds **{ NAME_MAX }** while **{ _POSIX_NO_TRUNC }** is in effect.
ENOENT	Either a component of the path prefix, or the file referred to by *path* does not exist or is a null pathname.
ENOLINK	*path* 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 the process does not possess the privilege PRIV_FILE_OWNER.
EPERM	The process does not possess the privilege PRIV_FILE_AUDIT and is attempting to set the **FAF_PUBLIC** flag.
EROFS	The file referred to by *path* resides on a read-only file system.

fsetfattrflag( ) fails and the file mode is unchanged if:

EACCES	The calling process does not own *fildes*. To override this restriction, the calling process may assert the PRIV_FILE_OWNER privilege.
EACCES	Write access is denied on *fildes*. To override this restriction, the calling process may assert the PRIV_FILE_MAC_WRITE privilege.
EINVAL	*fildes* is not a valid pathname. When setting **FAF_MLD**, *fildes* must refer to an empty directory.
EBADF	*fildes* is not an open file descriptor
EIO	An I / O error occurred while or writing to the file system.

EINTR	A signal was caught during execution of the **fsetfattrflag( )** function.
EPERM	The process does not possess the privilege PRIV_FILE_AUDIT and is attempting to set the **FAF_PUBLIC** flag.
EROFS	The file referred to by *fildes* resides on a read-only file system.

SEE ALSO

setfattrflag(1TSOL), **getfattrflag**(1TSOL)

Trusted Solaris Developer's Guide

NAME

getfpriv, fgetfpriv, setfpriv, fsetfpriv – return or set a privilege set associated with a file.

SYNOPSIS

int getfpriv(char **path*, **priv_ftype_t** *type*, **priv_set_t** **priv_set***)**

int setfpriv(char **path*, **priv_op_t** *op*, **priv_ftype_t** *type*, **priv_set_t** **priv_set)*

int fgetfpriv(int *fd*, **priv_ftype_t** *type*, **priv_set_t** **priv_set***)**

int fsetfpriv(int *fd*, **priv_op_t** *op*, **priv_ftype_t** *type*, **priv_set_t** **priv_set***)**

DESCRIPTION

Set or get privileges of the file that is named by *path* or referred to by *fd*. **fgetfpriv( )** and **fsetfpriv( )** function exactly like **getfpriv( )** and **setfpriv( )** respectively, except that they require an open reference to a file as their argument.

getfpriv( ) copies the privilege set indicated by *type* and associated with the named file into the address specified by *priv_set*. Values for *type* are:

PRIV_FORCED	The forced privilege set.
PRIV_ALLOWED	The allowed privilege set.

MAC read permission is required for the named file unless the privilege FILE_MAC_READ is effective.

setfpriv( ) sets / modifies the privilege set (the target set) indicated by *type* and associated with the named file. Modification occurs according to the value of *op* and the privilege set specified by *priv_set* (the specified set). Values for *op* are:

PRIV_ON	Each privilege asserted in the specified set is asserted in the target set.
PRIV_OFF	Each privilege asserted in the specified set is cleared in the target set.
PRIV_SET	The target set is set exactly equal to the specified set.

Values for *type* are the same as those used for **getfpriv( ).**

In all cases, the privilege FILE_SETPRIV must be effective. In addition, only the owner of a file may change its privilege sets, unless the privilege FILE_OWNER is effective.

The invoking process must have MAC write permission for the named file (unless the privilege FILE_MAC_WRITE is effective). DAC write access is not required.

It is an error to attempt to assert a forced privilege if the corresponding allowed privilege is not present. For this reason, it is recommended that the allowed privilege set be modified first whenever both privilege sets are to be modified.

If the target set is the allowed set, all privileges cleared from the target set are also automatically cleared from the forced set.

The following macros are provided for manipulating such privilege sets: **PRIV_ASSERT** (*priv_set, priv_id*) asserts privilege *priv_id* in privilege set *priv_set*. **PRIV_ISASSERT** (*priv_set, priv_id*) is non-zero if privilege *priv_id* in privilege set *priv_set* is asserted, zero otherwise. **PRIV_FILL** (*priv_set*) initializes a privilege set *priv_set* to the full set. **PRIV_ISFULL** (*priv_set*) is nonzero if the privilege set *priv_set* is a inverse null set, zero otherwise. **PRIV_EMPTY** (*priv_set*) initializes a privilege set *priv_set* to the null set.

PRIV_ISEMPTY (priv_set) is nonzero if the privilege set *priv_set* is a null set, zero otherwise. **PRIV_CLEAR** (*priv_set*, *priv_id*) clears the privilege *priv_id* in the privilege set *priv_set*. **PRIV_ISSUBSET** (*set_a*, *set_b*) is non-zero if all privileges asserted in privilege set *set_a* are also asserted in privilege set *set_b* (i.e., if *set_a* is a subset of *set_b*). **PRIV_UNION** (*set_a*, *set_b*) stores the union of *set_a* and *set_b* in *set_b*. **PRIV_INTERSECT** (set_a, set_b) stores the intersection of *set_a* and *set_b* in *set_b*. **PRIV_INVERSE** (*priv_set*) stores the inverse of *priv_set* in *priv_set*. The behavior of these macros is undefined if *priv_id* is less than one or greater than **PRIV_NUMPRIV**.

Normally MAC read permission is required or the privilege FILE_MAC_READ must be effective for **getfpriv( )** to complete its operation successfully unless the named file is a pty pseudo-terminal. If the named file is a pseudo-terminal (/ dev / ptyp* or / dev / ttyp*) and the label of the process invoking **getfpriv( )** does not dominate the label of the named file and the privilege FILE_MAC_READ is not effective then **getfpriv( )** returns success but sets the privilege fields of *priv_set* to zero.

The information label of *path* or *fildes* is unchanged. The information label of the calling process is also unchanged.

RETURN VALUES

These routines return:

0 on success.

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

ERRORS

These routines fail and the target set is not modified if:

EINVAL an illegal or undefined value is supplied for *size* or *type*.

EFAULT *priv_set* refers to an invalid address.

Additionally, **getfpriv( )** and **setfpriv( )** fail if:

EACCES search permission is denied a component of *path*. To override this restriction, the calling process may assert the PRIV_FILE_DAC_SEARCH privilege and / or the PRIV_FILE_MAC_SEARCH privilege.

getfpriv( ) and **fgetfpriv( )** fail if:

EACCES MAC read permission is denied for the named file, and privilege FILE_MAC_READ is not effective.

ENOENT a component of the specified path does not exist.

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

ENAMETOOLONG
the length of the path argument exceeds **PATH_MAX**, oL a pathname component is longer than **NAME_MAX** while **POSIX_NO_TRUNC** is in effect.

setfpriv( ) and **fsetfpriv( )** fail and the target set is not modified if:

EACCES MAC write permission is denied for the named file, privilege FILE_MAC_WRITE is not effective, and the user's clearance dominates

	the sensitivity label of the file.
EINVAL	(1) the named file resides on a file system that does not support privileges (i.e., a file system other than NFS, TMPFS) or (2) an illegal or undefined value is supplied for *op.* Also if privilege FILE_MAC_WRITE is not effective.
EPERM	MAC write permission is denied for the named file, and the user's clearance does not dominate the label of the named file, or (2) FILE_SETPRIV is not effective, or (3) the effective uid does not match the owner of the named file and privilege FILE_OWNER is not effective.
EROFS	the named file resides on a read-only filesystem.

SEE ALSO **getppriv**(2TSOL) **setppriv**(2TSOL) **priv_macros**(5TSOL)

NAME

getfsattr, fgetfsattr – get filesystem securtity attributes

SYNOPSIS

#include <tsol/fsattr.h>

int getfsattr(char ∗*path*, **u_long** *type* **, void** ∗*buf_p*, **int** *len***)**

int fgetfsattr(int *fd*, **u_long** *type* **, void** ∗*buf_p*, **int** *len***)**

DESCRIPTION

getfsattr( ) returns the file-system security attributes of a mounted file system. *path* is the pathname of any file within the mounted file system. *type* is the type of attribute requested. Values for *type* are:

FSA_ACLCNT	The file system access ACL count.
FSA_ACL	The file system access ACL.
FSA_DFACLCNT	The file system default ACL count.
FSA_DFACL	The file system default ACL.
FSA_APRIV	The file system allowed privilege set.
FSA_FPRIV	The file system forced privilege set.
FSA_LABEL	The file system CMW label.
FSA_AFLAGS	The file system attribute flags.
FSA_LBLRNG	The file system label range.
FSA_MLDPFX	The file system MLD prefix string.

buf_p is a pointer to a buffer to hold the requested attribute, and *len* is the buffer length.

fgetfsattr( ) returns the same information, but for an open file referred to by descriptor *fd*. *type*, *buf_p*, and *len* are the same as for **getfsattr()**.

The information label of *path* or *fd* is unchanged. The information label of the calling process is also unchanged.

RETURN VALUES

getfsattr( ) and **fgetfsattr( )** return:

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

ERRORS

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

EACCES	Search permission is denied for a component of the path prefix of *path*. To override this restriction, the calling process may assert the PRIV_FILE_DAC_SEARCH privilege and / or the PRIV_FILE_MAC_SEARCH privilege.
EFAULT	*buf_p* or *path* points to an invalid address.
EINVAL	The requested attributed is not set.
EIO	An I / O error occurred while reading from the filesystem.

ELOOP	Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of the path argument exceeds {PATH_MAX}.
	A pathname component is longer than {NAME_MAX} (see **sysconf**(2V)) while {_POSIX_NO_TRUNC} is in effect (see **pathconf**(2V)).
ENOENT	The file referred to by *path* does not exist.
ENOTDIR	A component of the path prefix of *path* is not a directory.

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

EBADF	*fd* is not a valid open file descriptor.
EFAULT	*buf_p* points to an invalid address.
EINVAL	*fd* refers to a socket, not a file; or the requested attribute is not set.
EIO	An I/O error occurred while reading from the file system.

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

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

setgroups( ) sets the supplementary group access list of the calling process from the array of group IDs specified by *grouplist*. The number of entries, specified by *ngroups*, cannot be greater than **NGROUPS_MAX**. The calling process must have **PRIV_PROC_SETID** in its set of effective privileges to set new groups. If **PRIV_PROC_SETID** is not in the effective privilege set, the operation fails and sets **errno** to **EPERM**.

RETURN VALUES

Upon successful completion, **getgroups** returns the number of supplementary group IDs set for the calling process; **setgroups** returns the value **0**. Upon failure, these functions return –**1** and set **errno** to indicate the error.

ERRORS

getgroups will fail if this condition is true:

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

setgroups will fail if either of these conditions is true:

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

EPERM The calling process does not have the **PRIV_PROC_SETID** privilege.

Either call will fail if this condition is true:

EFAULT A referenced part of the array to which *grouplist* points is an illegal address.

SUMMARY OF TRUSTED SOLARIS CHANGES

To set new groups, the calling process must have **PRIV_PROC_SETID** in its set of effective privileges.

SEE ALSO

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

NAME

getmldadorn, fgetmldadorn – get file system multilevel directory adornment

SYNOPSIS

#include <tsol/mld.h>

int getmldadorn(char **path_name*, **char** *adorn_buf***[MLD_ADORN_MAX]**

int fgetmldadorn(int *fd*, **char** *adorn_buf***[MLD_ADORN_MAX]**

DESCRIPTION

getmldadorn( ) returns the MLD adornment of the file system on which *path_name* resides. *path_name* is the path name of any file within the mounted filesystem. *adorn_buf* is a pointer to a buffer of at least MLD_ADORN_MAX bytes in which the null-terminated MLD adornment is returned.

fgetmldadorn( ) returns the same information about an open file referred to by descriptor *fd*.

The information label of *path_name* or *fd* is unchanged. The information label of the calling process is also unchanged.

RETURN VALUES

getmldadorn( ) and **fgetmldadorn( )** return:

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

ERRORS

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

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

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

EBADF	*fd* is not a valid open file descriptor.
EFAULT	*adorn_buf* points to an invalid address.
EINVAL	*fd* refers to a socket, not a file.
EIO	An I / O error occurred while reading from the file system.

WARNINGS

If the filesystem of the *fd* is not a CFS filesystem, no error is returned, and a zero-length string is returned in the *adorn_buf* buffer.

SEE ALSO

fgetsldname(2TSOL), **getsldname**(2TSOL)

NAME

getmsgqcmwlabel, getshmcmwlabel, getsemcmwlabel - Get the CMW labels associated with System V IPC structures

SYNOPSIS

#include <sys/tsol/ipcl.h>

cc [*flag* ...] *file* ... **–ltsol** [*library* ...]

int getmsgqcmwlabel(int *msgqid*, **bclabel_t** **clabel*)

int getshmcmwlabel(int *shmid*, **bclabel_t** **clabel*)

int getsemcmwlabel(int *semid*, **bclabel_t** **clabel*)

DESCRIPTION

These functions return the value of the CMW labels associated with message queues, shared memory, and semaphore structures.

getmsgqcmwlabel( ) returns the CMW label for the message queue identified by *msgqid* into the label buffer to which *clabel* points. The information label portion of the CMW label is undefined for message queues; therefore the sensitivity label portion may have to be extracted using **getcsl**(3TSOL) in order to be useful. Information labels are stored on individual messages and may be returned with **msgrcvl**(3TSOL).

getshmcmwlabel( ) returns the CMW label for the shared-memory segment identified by *shmid* into the label buffer to which *clabel* points.

getsemcmwlabel( ) returns the CMW label for the semaphore array identified by *semid* into the label buffer to which *clabel* points.

The calling process must have mandatory read access to the IPC or must have asserted the **PRIV_IPC_MAC_READ** privilege, and must have discretionary read access to the data structure or must have the **PRIV_IPC_DAC_READ** privilege in its set of effective privileges.

DESCRIPTION

Upon success, these functions return **0**. Upon failure, they return –**1** and set **errno** to indicate the error.

ERRORS

These functions will fail if any of these conditions is true:

EACCES	Read access is denied to the calling process, which does not have one or both of these privileges in its set of effective privileges: **PRIV_IPC_DAC_READ** and **PRIV_IPC_MAC_READ**.
EINVAL	*msgqid*,*semid*, or *shmid* is not a valid IPC object identifier.
EFAULT	*clabel* points to an illegal address.

SEE ALSO

msgget(2TSOL), **semget**(2TSOL), **shmget**(2TSOL), **msgrcvl**(2TSOL)

NAME

getpattr, setpattr – Get / set process attribute flags

SYNOPSIS

#include <tsol/pattr.h>

int getpattr(pattr_type_t *type*, **pattr_flag_t** *∗value*);

int setpattr(pattr_type_t *type*, **pattr_flag_t** *value***);**

DESCRIPTION

Process attribute flags are a set of flags that describe additional attributes that the process has. Each flag in the set is separately addressable although all flags share the **getpattr** and the **setpattr** system call interfaces. Likewise, each flag in the set has its own protection policy although all flags use the same protection mechanism. In the set are seven types of flags, specified in **<tsol/pattr.h>**, and addressed by the *type* argument. These are the values for *type*:

PAF_TRUSTED_PATH	Trusted path flag
PAF_PRIV_DBG	Privilege debugging flag
PAF_TOKMAPPER	Network token mapping process flag
PAF_LABEL_VIEW	Label view flags
PAF_LABEL_XLATE	Label translation flags
PAF_DISKLESS_BOOT	Part of diskless boot flag
PAF_SELAGNT	Part of selection agent flag
PAF_PRINT_SYSTEM	Part of trusted printing system flag

A description of each type of process attribute flag follows:

Trusted path flag

This one-bit flag marks a trusted path process. This flag can be viewed and cleared, but cannot be set. In other words, the call to **setpattr**(**PAF_TRUSTED_PATH, 1**) will always fail. A process inherits the trusted path flag from its parent process. The **init** process receives the trusted path flag from the system. A user session creator, such as **login**, clears this flag before starting a user session.

Privilege debugging flag

This one-bit flag indicates that the process is in privilege-debugging mode—a process-operation mode in which privilege requirement is logged but not enforced. This flag can be viewed or cleared, but cannot be set except by a trusted path process.

Network token mapping process flag

This one-bit flag, when set, identifies the process as the network token mapping process. The network token mapping process is exempt from network token mapping. This flag can be viewed and cleared, but cannot be set except by a trusted path process.

Label view flags

These two-bit flags support per-process label translation. These flags are viewable and modifiable without restriction.

Label translation flags These fifteen-bit flags support the GFI **FLAGS=** option in the **label_encodings** file. Only a trusted path process can view or modify these flags.

Part of diskless boot flag This one-bit flag identifies the process as taking part in diskless booting. This flag can be viewed and cleared, but cannot be set except by a trusted path process.

Part of selection agent flag This one-bit flag identifies the process as part of the “cut and paste” selection agent. This flag can be viewed and cleared, but cannot be set except by a trusted path process.

Part of trusted printing system flag This one-bit flag identifies the process as a member of the Trusted Printing System. This flag can be viewed and cleared, but cannot be set except by a trusted path process.

In short, these flag-related protection policies apply. Any process may view or clear any process attribute flag except the label translation flags; viewing or clearing the label translation flags requires that a process have the trusted path attribute. Any process may set label view flags; setting other flags requires that the setting process have the trusted path attribute.

getpattr() copies the *type* process flag of the calling process into the *pattr_flag_t* variable addressed by *value*. Only the lower *n* bits are copied, where *n* is the width of the flag. The higher bits are cleared.

setpattr( ) copies the lower *n* bits of *value* to the *type* process flag of the calling process, where *n* is the width of the selected process flag.

RETURN VALUES

Upon successful completion, the process returns a value of **0**. Otherwise, the process returns a value of −**1** and sets **errno** to indicate the error.

ERRORS

getpattr( ) may fail for one of these reasons:

EFAULT	The *value* argument points to a bad address.
EINVAL	The *type* argument is not one of the listed type constants.
EACCES	The calling process is not a trusted path process as required to view the *type* flag.

setpattr( ) may fail for one of these reasons:

EFAULT	The *value* argument points to a bad address.
EINVAL	The *type* argument is not one of the listed type constants.
EACCES	The calling process is not a trusted path process as required to modify the *type* flag.

SEE ALSO

pattr(1TSOL)

NAME

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

SYNOPSIS

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

pid_t getpid(void);

pid_t getpgrp(void);

pid_t getppid(void);

pid_t getpgid(pid_t *pid***);**

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

getpid( ) returns the process ID of the calling process.

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

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

getpgid( ) returns the process group ID of the process whose process ID is equal to *pid,* or the process group ID of the calling process, if *pid* is equal to zero. The calling process must have MAC read access to the target process. The calling process' real or effective user ID must match the real or saved user ID of the target process.

RETURN VALUES

Upon successful completion, all return the process group ID. On failure, **getpgid( ) returns a value of** (**pid_t**) –1 and sets **errno** to indicate the error.

ERRORS

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

EPERM	The process whose process ID is equal to *pid* is not in the same session as the calling process, and the implementation does not allow access to the process group ID of that process from the calling process.
ESRCH	There is no process with a process ID equal to *pid.* Or, the calling process does not have MAC read access to the target process, and does not have **PRIV_PROC_MAC_READ** overriding privilege. Or, the calling process' real or effective user ID does not match the real or saved user ID of the target process, and does not have **PRIV_PROC_OWNER** overriding privilege.

SUMMARY OF TRUSTED SOLARIS CHANGES

MAC and DAC policies are added to the **getpgid( )** command. To avoid covert channel issues, Trusted Solaris does not distinguish between failures due to policy and those due to nonexistence of the target process.

SEE ALSO

intro(2TSOL), **exec**(2TSOL), **fork**(2TSOL), **getsid**(2TSOL), **setpgid**(2), **setpgrp**(2), **signal**(3C)

NAME

getppriv, setppriv – Return or assign a privilege set associated with the invoking process

SYNOPSIS

#include <tsol/priv.h>

int getppriv(priv_ptype_t *type*, **priv_set_t*** *pset***);**

int setppriv(priv_op_t *op*, **priv_ptype_t** *type*, **priv_set_t*** *pset***);**

DESCRIPTION

getppriv( ) copies the *type* privilege set of the invoking process into the *pset* address. *type* may have one of four values, specified in **tsol/priv.h**:

PRIV_EFFECTIVE	The effective privilege set
PRIV_INHERITABLE	The inheritable privilege set
PRIV_PERMITTED	The permitted privilege set
PRIV_SAVED	The saved privilege set

setppriv( ) assigns or modifies the *type* privilege set (the target set) of the invoking process. Modification occurs according to the values of *op* and of the *pset* privilege set (the source set). *op* values are specified in **tsol/priv.h**:

PRIV_ON	Each privilege asserted in the source set is asserted in the target set.
PRIV_OFF	Each privilege asserted in the source set is cleared in the target set.
PRIV_SET	The target set is made exactly equal to the source set.

Values for *type* are the same as those for *type* in **getppriv**( ), exclusive of **PRIV_SAVED**.

If the target set is the permitted set, all privileges cleared from the target set are also cleared from the effective set. Any attempted assignment of a privilege cleared in the permitted set is always an error. Attempting to clear a privilege that is already cleared is not an error.

RETURN VALUES

getppriv( ) and **setppriv**( ) return **0** if successful. If not, they return –**1** and set **errno** to indicate the error.

ERRORS

getppriv( ) fails if either of these conditions prevails:

EINVAL	An illegal or undefined value was supplied for *type*.
EFAULT	*pset* refers to an invalid address.

setppriv( ) fails and the target set is not modified if any of these conditions prevails:

EINVAL	An illegal or undefined value is supplied for *type* or *op*.
EFAULT	*set* refers to an invalid address.
EINVAL	In a process privilege set, an attempt is made to assert a privilege that is cleared in the permitted set of the process.

SEE ALSO

getfpriv(2TSOL)C , **setfpriv**(2TSOL)C , **priv_to_str**(3TSOL)C , **priv_set_to_str**(3TSOL)C , **str_to_priv**(3TSOL)C , **str_to_priv_set**(3TSOL)C , **priv_macros**(5TSOL)

NAME

getrlimit, setrlimit – control maximum system resource consumption

SYNOPSIS

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

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

int setrlimit(int *resource***, const struct rlimit** ∗*rlp***);**

DESCRIPTION

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

Each call to either **getrlimit( )** or **setrlimit( )** identifies a specific resource to be operated upon as well as a resource limit. A resource limit is a pair of values: one specifying the current (soft) limit, the other a maximum (hard) limit. Soft limits may be changed by a process to any value that is less than or equal to the hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or equal to the soft limit. Only a process that has the **PRIV_SYS_CONFIG** privilege can raise a hard limit. Both hard and soft limits can be changed in a single call to **setrlimit( )** subject to the constraints described above. Limits may have an "infinite" value of **RLIM_INFINITY**. *rlp* is a pointer to **struct rlimit** that includes the following members:

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

rlim_t is an arithmetic data type to which objects of type **int**, **size_t**, and **off_t** can be cast without loss of information.

The possible resources, their descriptions, and the actions taken when the current limit is exceeded are summarized in the table below:

RLIMIT_CORE	The maximum size of a core file in bytes that may be created by a process. A limit of **0** will prevent the creation of a core file. The writing of a core file will terminate at this size.
RLIMIT_CPU	The maximum amount of CPU time in seconds used by a process. This is a soft limit only. **SIGXCPU** is sent to the process. If the process is holding or ignoring **SIGXCPU**, the behavior is scheduling class defined.
RLIMIT_DATA	The maximum size of a process's heap in bytes. **brk**(2) will fail with **errno** set to **ENOMEM.**
RLIMIT_FSIZE	The maximum size of a file in bytes that may be created by a process. A limit of **0** will prevent the creation of a file. **SIGXFSZ** is sent to the process. If the process is holding or ignoring **SIGXFSZ**, continued attempts to increase the size of a file beyond the limit will fail with **errno** set to **EFBIG.**

RLIMIT_NOFILE	One more than the maximum value that the system may assign to a newly created descriptor. This limit constrains the number of file descriptors that a process may create.
RLIMIT_STACK	The maximum size of a process's stack in bytes. The system will not automatically grow the stack beyond this limit. **SIGSEGV** is sent to the process. If the process is holding or ignoring **SIGSEGV**, or is catching **SIGSEGV** and has not made arrangements to use an alternate stack (see **sigaltstack**(2)), the disposition of **SIGSEGV** will be set to **SIG_DFL** before it is sent.
RLIMIT_VMEM	The maximum size of a process's mapped address space in bytes. **brk**(2) and **mmap**(2) functions will fail with **errno** set to **ENOMEM.** In addition, the automatic stack growth will fail with the effects outlined above.

Because limit information is stored in the per-process information, the shell builtin **ulimit** command must directly execute this system call if it is to affect all future processes created by the shell.

The value of the current limit of the following resources affect these implementation defined parameters:

Limit	Implementation Defined Constant
RLIMIT_FSIZE	**FCHR_MAX**
RLIMIT_NOFILE	**OPEN_MAX**

RETURN VALUES

Upon successful completion, the function **getrlimit( )** returns a value of 0; otherwise, it returns a value of –1 and sets **errno** to indicate an error.

ERRORS

Under the following conditions, the functions **getrlimit( )** and **setrlimit( )** fail and set **errno** to:

EFAULT	*rlp* points to an illegal address.
EINVAL	An invalid *resource* was specified; or in a **setrlimit( )** call, the new **rlim_cur** exceeds the new **rlim_max**.
EPERM	The limit specified to **setrlimit( )** would have raised the maximum limit value, and the calling process does not have the **PRIV_SYS_CONFIG** privilege.

SUMMARY OF TRUSTED SOLARIS CHANGES

The calling process must have the **PRIV_SYS_CONFIG** privilege in order to increase a hard resoure limit.

SEE ALSO

brk(2), **open**(2TSOL), **sigaltstack**(2), **malloc**(3C), **signal**(3C), **signal**(5)

NOTES

RLIMIT_STACK:

Within a process **setrlimit()**, will increase the limit on the size of your stack, but will not move current memory segments to allow for that growth. Therefore, to guarantee that the process stack can grow to the limit, you must alter the limit prior to the execution of the process in which the new stack size is to be used.

NAME

getsid, setsid – get or set session ID

SYNOPSIS

#include <sys/types.h>

pid_t getsid(pid_t *pid***);**

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

pid_t setsid(void);

MT-LEVEL

setsid( ) is Async-Signal-Safe

DESCRIPTION

The function **getsid( )** returns the session ID of the process whose process ID is equal to *pid.* If *pid* is equal to **(pid_t)0**, **getsid( )** returns the session ID of the calling process. The calling process must have MAC read access to the target process. The calling process' real or effective user ID must match the real or saved user ID of the target process.

If the calling process is not already a process group leader, **setsid( )** sets the process group ID and session ID of the calling process to the process ID of the calling process, and releases the process's controlling terminal.

See **intro**(2) for more information on process groups and controlling terminals.

RETURN VALUES

Upon successful completion, **getsid( )** and **setsid( )** return the session ID of the specified process. Otherwise, **getsid( )** returns a value of **(pid_t)**–**1** and sets **errno** to indicate an error, and **setsid( )** returns a value of –1 and sets **errno** to indicate the error.

ERRORS

Under the following conditions, **getsid( )** fails and sets **errno** to:

EPERM The process whose process ID is equal to *pid* is not in the same session as the calling process, and the implementation does not allow access to the session ID of that process from the calling process.

ESRCH There is no process with a process ID equal to *pid.* Or, the calling process does not have MAC read access to the target process, and does not have **PRIV_PROC_MAC_READ** overriding privilege. Or, the calling process' real or effective user ID does not match the real or saved user ID of the target process, and does not have **PRIV_PROC_OWNER** overriding privilege.

setsid( ) will fail and return an error if the following is true:

EPERM The calling process is already a process group leader, or there are processes other than the calling process whose process group ID is equal to the process ID of the calling process.

SUMMARY OF TRUSTED SOLARIS CHANGES

MAC and DAC policies are added to the **getsid( )** command.

SEE ALSO

intro(2TSOL), **exec**(2TSOL), **fork**(2TSOL), **getpid**(2TSOL), **setpgid**(2TSOL)

WARNINGS

A call to **setsid( )** by a process that is a process group leader will fail. A process can become a process group leader by being the last member of a pipeline started by a job control shell. Thus, a process that expects to be part of a pipeline, and that calls **setsid( )**, should always first fork; the parent should exit and the child should call **setsid( )**. This will ensure that the calling process will work reliably when started by both job control shells and non-job control shells.

NAME

getsldname, fgetsldname – Get file system single-level directory name

SYNOPSIS

#include <tsol/label.h>

int getsldname(char **path_name*, **bslabel_t** **slabel_p* , **char** **name_buf*, **const int** *length***)**

int fgetsldname(const int *fd*, **const bslabel_t** **slabel_p*, **char** **name_buf* , **const int** *length***)**

AVAILABILITY

SUNWtsolu

DESCRIPTION

getsldname( ) returns the SLD name associated with the sensitivity label to which *slabel_p* refers within the context of the file system on which *path_name* resides. *path_name* is the path name of any multilevel directory within the mounted filesystem. *name_buf* is a pointer to a buffer of at least **SLD_NAME_MAX** bytes.

fgetsldname( ) returns the SLD name associated with the sensitivity label to which *slabel_p* refers if the MLD to which descriptor *fd* refers was opened by the directory name (not by the fully adorned, multilevel directory name.) If the MLD to which descriptor *fd* refers was opened using the fully adorned, multilevel directory name, **fgetsldname** returns the MLD and the SLD name associated with the sensitivity label to which *slabel_p* refers.

If it does not exist, the single-level directory that corresponds to *slabel_p* is created with the attributes of the parent multilevel directory, the specified sensitivity label, and an ADMIN_LOW information label. If the sensitivity label of the calling process is equal to *slabel_p*, no additional privileges are needed. If the sensitivity label of the calling process is strictly dominated by *slabel_p*, the calling process may assert the **PRIV_FILE_UPGRADE_SL** privilege to create the directory. Otherwise, the calling process may assert the **PRIV_FILE_DOWNGRADE_SL** privilege to create the directory.

RETURN VALUES

Upon success, these functions return **0**. Upon failure, they return –**1** and set **errno** to indicate the error.

ERRORS

getsldname fails if any of these conditions is true:

EACCES Search permission is denied for a component of the path prefix of *path_name*. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_SEARCH** and **PRIV_FILE_MAC_SEARCH**.

The single-level directory specified does not exist, the system is configured to require write access to create a single-level directory, and the calling process does not have discretionary write access to *path_name*. To override this restriction, the calling process may assert the **PRIV_FILE_DAC_WRITE** privilege.

EFAULT *name_buf*, *path_name*, or *slabel_p* points to an invalid address.

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

	system
ELOOP	Too many symbolic links were encountered in translating *path_name*.
ENAMETOOLONG	The length of the path argument exceeds {**PATH_MAX**}.
	A pathname component is longer than {**NAME_MAX**} [see **sysconf**(2V)] while {**_POSIX_NO_TRUNC**} is in effect. [See **pathconf**(2V).]
ENOENT	The file to which *path_name* refers does not exist.
ENOTDIR	A component of the path prefix of *path_name* is not a directory.
EPERM	The SLD that corresponds to *slabel_p* does not exist and one of these conditions is true: the sensitivity label of the calling process is strictly dominated by *slabel_p* and the calling process has not asserted the **PRIV_FILE_DOWNGRADE** privilege; the sensitivity label of the calling process is not dominated by *slabel_p* and the calling process has not asserted the **PRIV_FILE_DOWNGRADE_SL** privilege.

fgetsldname fails if any of these conditions is true:

EBADF	*fd* is not a valid open file descriptor.
EFAULT	*name_buf* or *slabel_p* points to an invalid address.
EINVAL	*fd* does not refer to a multilevel directory.
EIO	An I/O error occurred while reading from the file system.
EPERM	The SLD that corresponds to *slabel_p* does not exist and one of these conditions is true: the sensitivity label of the calling process is strictly dominated by *slabel_p* and the calling process has not asserted the **PRIV_FILE_UPGRADE_SL** privilege; the sensitivity label of the calling process is not dominated by *slabel_p* and the calling process has not asserted the **PRIV_FILE_DOWNGRADE_SL** privilege.

WARNINGS

If the file system that contains *path_name* or the object referred to by *fd* does not support MLDs, no error is returned and the first {**SLD_NAME_MAX**} bytes in the *name_buf* are cleared.

SEE ALSO

fgetmldadorn(2TSOL), **getmldadorn**(2TSOL)

NAME

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

SYNOPSIS

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

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

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

kill( ) sends a signal to a process or a group of processes specified by *pid*. The signal that is to be sent, specified by *sig*, is either one from the list given in **signal** [see **signal**(5)], or **0**. If *sig* is **0** (the null signal), error checking is performed but no signal is actually sent. This method can be used to check the validity of *pid*.

The sending process must have MAC write access to the receiving processes. The real or effective user ID of the sending process must match the real or saved [from **exec**(2)] user ID of the receiving process unless the sending process has the **PRIV_PROC_OWNER** effective privilege, or *sig* is **SIGCONT** and the sending process has the same session ID as the receiving process.

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

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

If *pid* is **0**, *sig* will be sent to all processes excluding special processes [see **intro**(2)] whose process group ID is equal to the process group ID of the sender.

If *pid* is **(pid_t)**–**1** and the sender does not have **PRIV_PROC_OWNER** in its effective privilege set, *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 sender does not have **PRIV_PROC_OWNER** in its effective privilege set, *sig* will be sent to all processes excluding special processes.

RETURN VALUES

Upon successful completion, **kill** returns a value of **0**. Upon failure, **kill** returns a value of –**1** is returned and sets **errno** to indicate the error.

ERRORS

kill will fail and no signal will be sent if any of these conditions is true:

EINVAL *sig* is not a valid signal number.

EPERM The calling process failed in MAC write access to the receiving process and does not have **PRIV_PROC_MAC_WRITE** overriding privilege.

sig is **SIGKILL** and *pid* is **(pid_t)1**. (That is, the calling process does not have permission to send the signal to any of the processes specified by *pid*).

The effective user of the calling process does not match the real or saved user and the sending process does not have **PRIV_PROC_OWNER** privilege, and the calling process is not sending **SIGCONT** to a process that shares the same session ID.

ESRCH No process or process group can be found corresponding to that specified by *pid*.

SUMMARY OF TRUSTED SOLARIS CHANGES

Process MAC write policy and the process owner policy is checked.

SEE ALSO

kill(1), **intro**(2TSOL), **exec**(2TSOL), **getpid**(2TSOL), **getsid**(2TSOL), **setpgrp**(2), **sigaction**(2), **sigsend**(2TSOL), **signal**(3C), **signal**(5)

NOTES

sigsend(2) is a more versatile way to send signals to processes.

NAME

link – Link to a file

SYNOPSIS

#include <unistd.h>

int link(const char **existing*, **const char** **new***);**

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

link( ) creates a new link (directory entry) for the existing file and increments its link count by one. *existing* points to an existing file. *new* points to a new directory entry to be created.

For creation of 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. A calling process that has asserted the **PRIV_SYS_CONFIG** privilege may make multiple links to a directory.

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

RETURN VALUES

Upon successful completion, **link** returns **0**. Upon failure, **link** returns –**1** and sets **errno** to indicate the error.

ERRORS

link( ) will fail and no link will be created if any of these conditions is true:

EACCES	A component of either path prefix denies search permission. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_MAC_SEARCH** and **PRIV_FILE_DAC_SEARCH**.
	The requested link requires writing in a directory with a mode that denies write permission. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_MAC_WRITE** and **PRIV_FILE_DAC_WRITE**.
	The calling process needs both mandatory read and write access to *existing* and does not have that combination. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_MAC_READ** and **PRIV_FILE_MAC_WRITE**.
EDQUOT	The directory in which the entry for the new link is being placed cannot be extended because the user's quota of disk blocks on that file system has been exhausted.
EEXIST	The link named by *new* exists.
EFAULT	*existing* or *new* points to an illegal address.
EINTR	A signal was caught during the **link** function.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMLINK	The maximum number of links to a file would be exceeded.

EMULTIHOP	Components of *existing* or *new* require hopping to multiple remote machines and the file system type does not allow it.
ENAMETOOLONG	The length of the *existing* or *new* argument exceeds {**PATH_MAX**}, or the length of a *existing* or *new* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	*existing* or *new* is a null path name.
	A component of either path prefix does not exist.
	The file named by *existing* does not exist.
ENOLINK	*existing* or *new* points to a remote machine but the link to that machine is no longer active.
ENOSPC	The directory that would contain the link cannot be extended.
ENOTDIR	A component of either path prefix is not a directory.
EPERM	The file named by *existing* is a directory and the calling process has not asserted the **PRIV_SYS_CONFIG** privilege.
EROFS	The requested link requires writing in a directory on a read-only file system.
EXDEV	The link named by *new* and the file named by *existing* are on different logical devices (file systems).

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

If *existing* is a directory, the calling process must assert the **PRIV_SYS_CONFIG** privilege.

SEE ALSO

symlink(2TSOL), **unlink**(2TSOL)

NAME

llseek – Move the extended read / write file pointer

SYNOPSIS

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

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

DESCRIPTION

llseek( ) sets the 64-bit extended file pointer associated with the open file descriptor specified by *fildes* as follows:

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

On success, **llseek** returns the resulting pointer location, measured in bytes from the beginning of the file.

Discretionary access checks have already been performed when *fildes* was opened.

Most mandatory access checks have already been performed when *fildes* was opened. If *fildes* is open for writing, a check is made that the calling process has mandatory read access in case *fildes* is open for a write-up. The calling process may assert the **PRIV_FILE_MAC_READ** prvilege to bypass this check. If mandatory read access is not granted, this system call succeeds; but offset data is not returned.

The information labels of *fildes* and of the calling process are unchanged.

RETURN VALUES

Upon successful completion, **llseek** returns the resulting file pointer. Remote file descriptors are the only ones that allow negative file pointers. Upon failure, **llseek** returns **–1** and sets **errno** to indicate the error.

ERRORS

llseek fails and the file pointer remains unchanged if any of these conditions is true:

EBADF	*fildes* is not an open file descriptor.
EINVAL	*whence* is not **SEEK_SET**, **SEEK_CUR**, nor **SEEK_END**.
EINVAL	*offset* is not a valid offset for this file-system type.
EINVAL	*fildes* is not a remote file descriptor, and the resulting file pointer would be negative.
ESPIPE	*fildes* is associated with a pipe or FIFO.

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

LIMITATIONS

Although each file has a 64-bit file pointer associated with it, existing file-system types do not support the full range of 64-bit offsets. In particular, nondevice files remain limited to offsets of less than two gigabytes. Device drivers may support offsets of up to 1024 gigabytes for device special files.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

Discretionary access checks have already been performed when *fildes* was opened.

Most mandatory access checks have already been performed when *fildes* was opened. The calling process may assert the **PRIV_FILE_MAC_READ** prvilege to perform a write-up.

The information labels of *fildes* and of the calling process are unchanged.

SEE ALSO

creat(2TSOL), **dup**(2), **fcntl**(2TSOL), **lseek**(2TSOL), **open**(2TSOL)

NAME

lseek – Move the read / write file pointer

SYNOPSIS

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

off_t lseek(int *fildes***, off_t** *offset***, int** *whence***);**

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

lseek( ) sets the file pointer associated with the open file descriptor specified by *fildes* as follows:

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

On success, **lseek** returns the resulting pointer location measured in bytes from the beginning of the file. Note that if *fildes* is a remote file descriptor and *offset* is negative, **lseek** returns the file pointer even if it is negative.

lseek allows the file pointer to be set beyond the existing data in the file. If data 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 of value 0 until data is written into the gap.

Discretionary access checks have already been performed when *fildes* was opened.

Most mandatory access checks have already been performed when *fildes* was opened. If *fildes* is open for writing, a check is made that the calling process has mandatory read access in case *fildes* is open for a write-up. The calling process may assert the **PRIV_FILE_MAC_READ** privilege to bypass this check. If mandatory read access is not granted, this system call succeeds; but offset data is not returned.

The information labels of *fildes* and of the calling process are unchanged.

RETURN VALUES

Upon successful completion, **lseek** returns the resulting file pointer. Remote file descriptors are the only ones that allow negative file pointers. Upon failure, **lseek** returns –**1** and sets **errno** to indicate the error.

ERRORS

lseek( ) fails and the file pointer remains unchanged if any of these conditions is true:

EBADF	*fildes* is not an open file descriptor.
EINVAL	*whence* is not **SEEK_SET**, **SEEK_CUR**, nor **SEEK_END**.
EINVAL	*fildes* is not a remote file descriptor, and the resulting file pointer would be negative.
ESPIPE	*fildes* is associated with a pipe or FIFO.

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

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

Discretionary access checks have already been performed when *fildes* was opened.

Most mandatory access checks have already been performed when *fildes* was opened. The calling process may assert the **PRIV_FILE_MAC_READ** privilege to perform a write-up.

The information labels of *fildes* and of the calling process are unchanged.

SEE ALSO

creat(2TSOL), **dup**(2), **fcntl**(2TSOL), **open**(2TSOL), **read**(2TSOL), **write**(2TSOL)

NOTES

In multithreaded programs, using **lseek( )** in conjunction with a **read( )** or **write( )** on a file descriptor shared among more than one thread is not an atomic operation. To ensure atomicity, use **pread( )** or **pwrite( ).**

NAME

mkdir – Make a directory

SYNOPSIS

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

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

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

mkdir( ) creates a new directory with the path name to which *path* points. The mode of the new directory is initialized from *mode*. [See **chmod**(2TSOL) for values of mode.] The protection part of the *mode* argument is modified by the file-creation mask of the process. [See **umask**(2).]

The directory's owner ID is set to effective user ID of the process. The directory's group ID is set to effective group ID of the process; 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 except for 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. This system call will not create a directory in a multilevel directory. Single-level directories are automatically created as needed during path-name lookup and the **getsldname**(2TSOL) system call.

Trusted Solaris distinguishes multilevel directories from regular directories by their names. A multilevel directory has the adornment MLD; a single-level directory has the adornment SLD and a number. Use the **mldpwd** command within a multilevel directory to see the adorned names of the multilevel directory and the single-level directories. For example, executing the **mldpwd** command within the *user_name* home directory shows this output:

/export/home/.MLD.*user_name***/.SLD.2**

Use the **mldrealpath** command to see the adorned name for a file or directory within a multilevel directory. For example, **mldrealpath file.c** shows this output:

/export/home/.MLD.*user_name***/.SLD.2/file.c**

The new directory is created with its sensitivity label set to the sensitivity label of the calling process.

If the new directory's containing directory has a default access control list (ACL), the default and access ACLs of the new directory are set to the default ACL of the containing directory.

The information label of the new directory is set to ADMIN_LOW.

RETURN VALUES

Upon successful completion, **mkdir** returns **0**. Upon failure, **mkdir** returns –**1** and sets **errno** to indicate the error.

ERRORS

mkdir( ) fails and creates no directory if any of these conditions is true:

EACCES	Either a component of the path prefix denies search permission or write permission is denied on the parent directory of the directory to be created. To override these restrictions, the calling process may assert one or more of these privileges: **PRIV_FILE_DAC_SEARCH**, **PRIV_FILE_MAC_SEARCH**, **PRIV_FILE_DAC_WRITE**, and **PRIV_FILE_MAC_WRITE**.
EINVAL	An attempt was made to create a directory at a sensitivity label outside the range of the file system.
EDQUOT	The directory in which 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. The user's quota of inodes on the file system in which the file is being created has been exhausted.
EEXIST	The named file already exists.
EFAULT	*path* points to an illegal address.
EIO	An I/O error has occurred while accessing the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMLINK	The maximum number of links to the parent directory would be exceeded.
EMULTIHOP	Components of *path* require hopping to multiple remote machines, and the file system type does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	A component of the path prefix does not exist or is a null path name.
ENOLINK	*path* points to a remote machine but the link to that machine is no longer active.
ENOSPC	No free space is available on the device containing the directory.
ENOTDIR	A component of the path prefix is not a directory.
EROFS	The path prefix resides on a read-only file system.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

chmod(2TSOL), **mknod**(2TSOL), **umask**(2), **stat**(5)

NAME

mknod – Make a directory or a special or an ordinary file

SYNOPSIS

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

int mknod(const char **path*, **mode_t** *mode*, **dev_t** *dev*);

DESCRIPTION

mknod( ) creates a new file with the path name to which *path* points. The file type and permissions of the new file are initialized from *mode*. This system call will not create an object in a multilevel directory. Single-level directories are automatically created during path-name lookup and **getsldname**(2TSOL).

The new object is created with its sensitivity label set to the sensitivity label of the calling process. If the containing directory has a default access control list (ACL), the ACL is copied to the new object as its access ACL.

The information label of the final component of *path* is set to ADMIN_LOW.

The file type is specified in *mode* by the **S_IFMT** bits, which must be set to one of these values:

S_IFIFO	FIFO special
S_IFCHR	character special
S_IFDIR	directory
S_IFBLK	block special
S_IFREG	ordinary file

The file access permissions are specified in *mode* by the 0007777 bits, and may be constructed by an OR of these values:

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

The owner ID of the file is set to the effective user ID of the process. The group ID of the file is set to the effective group ID of the process. However, if the **S_ISGID** bit is set in the parent directory, then the group ID of the file is inherited from the parent. If the group ID of the new file does not match the effective group ID or one of the supplementary group IDs, the **S_ISGID** bit is cleared. To override this restriction,the calling process may assert the **PRIV_FILE_SETID** privilege.

If the file is not a directory, mode bit 01000 (save text image on execution) is cleared. The calling process may assert the **PRIV_SYS_CONFIG** privilege to override this restriction.

The access permission bits of *mode* are modified by the file-mode creation mask of the process: all bits set in the file-mode creation mask of the process are cleared. [See **umask**(2).] If *mode* indicates a block or character special file, *dev* is a configuration-dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored. [See **makedev**(3C).]

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

If *path* is a symbolic link, it is not followed.

RETURN VALUES

Upon successful completion, **mknod** returns **0**. Upon failure, **mknod** returns –**1** and sets **errno** to indicate the error.

ERRORS

mknod( ) fails and creates no new file if any of these conditions is true:

EACCESS	The calling process does not have search access to all directories in the object's path. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_SEARCH** and **PRIV_FILE_MAC_SEARCH**. The calling process does not have write access to the object's containing directory. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_WRITE** and **PRIV_FILE_MAC_WRITE**.
EDQUOT	The directory in which 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 user's quota of inodes on the file system in which the file is being created has been exhausted.
EEXIST	The named file exists.
EFAULT	*path* points to an illegal address.
EINTR	A signal was caught during the **mknod** function.
EINVAL	*dev* is invalid.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines, and the file system type does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	A component of the path prefix does not exist or is a null path

	name.
ENOLINK	*path* points to a remote machine but the link to that machine is no longer active.
ENOSPC	No space is available.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	The value in *mode* is not a FIFO and the calling process has not asserted the **PRIV_SYS_DEVICES** privilege.
EROFS	The directory in which the file is to be created is located on a read-only file system.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

The new object is created with its sensitivity label set to the sensitivity label of the calling process. If the containing directory has a default access control list (ACL), the ACL is copied to the new object as its access ACL.

SEE ALSO

chmod(2TSOL), **exec**(2TSOL), **mkdir**(2TSOL), **umask**(2), **makedev**(3C), **mkfifo**(3C), **stat**(5)

NOTES

Normally, applications should use the **mkdir**(2) routine to make a directory because **mknod** may not establish directory entries for the directory itself (**.**) and its parent directory) (**..**), and special privileges are not required. Similarly, **mkfifo**(3C) should be used in preference to **mknod** in order to create FIFOs.

NAME

mount – Mount a file system

SYNOPSIS

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

int mount(const char **spec***, const char** **dir***, int** *mflag***, /* char** **fstype***,**
const char **dataptr***, int** *datalen* ***/ . . .);**

DESCRIPTION

mount( ) requests that a removable file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. *spec* and *dir* are pointers to path names. *fstype* is the file system type, which can be determined by the **sysfs**(2) function. If both the **MS_DATA** and **MS_FSS** flag bits of *mflag* are off, the file system type defaults to the root file system type. Only if either flag is on is *fstype* used to indicate the file system type.

If the **MS_DATA** flag is set in *mflag*, the system expects the *dataptr* and *datalen* arguments to be present. Together they describe a block of file-system-specific data at address *dataptr* of length *datalen*. This data is interpreted by file-system-specific code within the operating system and its format depends on the file system type. If a particular file system type does not require this data, *dataptr* and *datalen* should both be zero. Note that **MS_FSS** is obsolete and is ignored if **MS_DATA** is also set; but if **MS_FSS** is set and **MS_DATA** is not, *dataptr* and *datalen* are both assumed to be zero.

After a successful call to **mount**, all references to the file *dir* refer to the root directory on the mounted file system.

The low-order bit of *mflag* is used to control write permission on the mounted file system: if that flag is **1**, writing is forbidden; otherwise writing is permitted according to individual file accessibility.

The **mount** system call may be invoked for all file system types except *namefs* by a calling process with the **PRIV_SYS_MOUNT** privilege. For the *namefs* file system, the calling process must either be the owner of *dir* or assert the **PRIV_FILE_OWNER** privilege.

RETURN VALUES

Upon successful completion, **mount** returns a value of **0**. Upon failure, **mount** returns a value of –**1** and sets **errno** to indicate the error.

ERRORS

mount fails if any of these conditions is true:

EACCES Search permission is denied on a component of *spec* or *dir*. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_SEARCH** and **PRIV_FILE_MAC_SEARCH**.

Write permission is denied to the *namefs* file system specified in *dir*. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_WRITE** and **PRIV_FILE_MAC_WRITE**.

EBUSY	*dir* is currently mounted on, is someone's current working directory, or is otherwise busy.
	The device associated with *spec* is currently mounted.
EFAULT	*spec*, *dir*, or *datalen* points outside the allocated address space of the process.
EINVAL	The super block has an invalid magic number or the *fstype* is invalid.
ELOOP	Too many symbolic links were encountered in translating *spec* or *dir*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines but the file system type does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	None of the named files exists or the path name is null.
ENOTBLK	*spec* is not a block special device.
ENOTDIR	*dir* is not a directory.
EPERM	The calling process does not own *dir* and *dir* is type *namefs*. To override this restriction, the calling process may assert the **PRIV_FILE_OWNER** privilege.
	dir is not a file system of type *namefs* and the calling process has not asserted the **PRIV_SYS_MOUNT** privilege.
EREMOTE	*spec* is remote and cannot be mounted.
ENOLINK	*path* points to a remote machine but the link to that machine is no longer active.
ENXIO	The device associated with *spec* does not exist.
EROFS	*spec* is write protected and *mflag* requests write permission.
ENOSPC	The file system state in the super-block is not **FsOKAY** and *mflag* requests write permission.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access or ownership checks.

The **mount** system call may be invoked for all file system types except *namefs* by a calling process with the **PRIV_SYS_MOUNT** privilege. For the *namefs* file system, the calling process must either be the owner of *dir* or assert the **PRIV_FILE_OWNER** privilege.

SEE ALSO

mount(1MTSOL), **sysfs**(2), **umount**(2TSOL)

NAME

msgctl – Message-control operations

SYNOPSIS

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

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

DESCRIPTION

msgctl( ) provides a variety of message-control operations as specified by *cmd*. These *cmd*s are available:

IPC_STAT Place the current value of each member of the data structure associated with *msqid* into the structure to which *buf* points. The contents of this structure are defined in **intro**(2).

If it does not have discretionary read access to the data structure, the calling process must have **PRIV_IPC_DAC_READ** in its set of effective privileges. If it does not have mandatory read access to the data structure, the calling process must have **PRIV_IPC_MAC_READ** in its set of effective privileges.

IPC_SET Set the value of these members of the data structure associated with *msqid* to the corresponding value found in the structure to which *buf* points:

msg_perm.uid
msg_perm.gid
msg_perm.mode /* only access permission bits */
msg_qbytes

A process whose effective user ID does not match the value of **msg_perm.cuid** or **msg_perm.uid** must have the **PRIV_IPC_OWNER** privilege in its set of effective privileges. A process must have mandatory write access to the data structure or must have asserted the **PRIV_IPC_MAC_WRITE** privilege. Only a process with **PRIV_SYS_IPC_CONFIG** asserted can raise the value of **msg_qbytes**.

IPC_RMID Remove from the system the message-queue identifier specified by *msqid* and destroy the message queue and data structure associated with it. This *cmd* can be executed only by a process that has an effective user ID equal to that of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*, or has the **PRIV_IPC_OWNER** privilege asserted. A process must also have mandatory write access to the data structure or must have asserted the **PRIV_IPC_MAC_WRITE** privilege. *buf* is ignored.

RETURN VALUES

Upon successful completion, this call returns a value of **0**. Upon failure, the call returns a value of –**1** and sets **errno** to indicate the error.

ERRORS

msgctl( ) fails if any of these conditions is true:

EACCES *cmd* is **IPC_STAT,** operation permission is denied to the calling process [see **intro**(2TSOL)], and the calling process does not have the appropriate privilege(s) in its set of effective privileges.

EFAULT *buf* points to an illegal address.

EINVAL *msqid* is not a valid message queue identifier.

cmd is not a valid command.

cmd is **IPC_SET** and **msg_perm.uid** or **msg_perm.gid** is not valid.

EPERM *cmd* is **IPC_RMID** or **IPC_SET**, the discretionary and / or mandatory access checks failed, and the process did not have the appropriate override privilege asserted.

cmd is **IPC_SET**, an attempt is being made to increase to the value of **msg_qbytes**, and the process did not have the appropriate override privilege asserted.

EOVERFLOW *cmd* is **IPC_STAT** and *uid* or *gid* is too large to be stored in the structure to which *buf* points.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

intro(2TSOL), **msgget**(2TSOL), **msgop**(2TSOL)

NAME

msgget, msggetl - Get message queue

SYNOPSIS

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

int msgget(key_t *key***, int** *msgflg***);**

cc [*flag* ...] *file* ... **-ltsol** [*library* ...]
#include <sys/tsol/ipcl.h>

int msggetl(**key_t** *key*, **int** *msgflg*, **const bslabel_t** ∗*slabel*);

DESCRIPTION

A message queue is identified by a unique combination of key and sensitivity label. This qualification of keys by sensitivity labels allows applications that use message queues to be run at multiple process sensitivity labels without inadvertently sharing data.

msgget( ) returns the message-queue identifier associated with the union of *key* and the sensitivity label of the calling process.

msggetl( ) returns the message-queue identifier associated with the union of *key* and *slabel*. If the value of *slabel* does not match the sensitivity label of the calling process, then the effective privilege set of the process must contain **PRIV_IPC_MAC_READ** or **PRIV_IPC_MAC_WRITE**.

If discretionary read / write access as specified by the low-order 9 bits of *msgflg* is denied to the calling process, **msgget**( ) and **msggetl**( ) require one or both of these privileges: **PRIV_IPC_DAC_READ** and **PRIV_IPC_DAC_WRITE**.

A message-queue identifier and associated message queue and data structure [see **intro**(2)] are created for *key* if one of the following is true:

key is **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, respectively, to the effective user ID and effective group IDs 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.

The sensitivity label on the message-queue internal is set either to the sensitivity label of the process or to *slabel*, depending on which interface was used.

RETURN VALUES

Successful completion returns a nonnegative integer, namely a message-queue identifier. Failure returns a value of **–1** and sets **errno** to indicate the error.

ERRORS

msgget and **msggetl** fail if any of these conditions is true:

EACCES	A semaphore-structure identifier exists for the union of key and sensitivity label, but operation permission [see intro(2)] as specified by the low-order 9 bits of *semflg* would not be granted; or the sensitivity label check did not pass, and the calling process does not have the appropriate privilege override(s) in its set of effective privileges.
EEXIST	A message queue identifier exists for *key* but (*msgflg*&**IPC_CREAT**) and (*msgflg*&**IPC_EXCL**) are both true.
EFAULT	*slabel* points to an illegal address.
EINVAL	The label to which *slabel* points is not a valid sensitivity label.
ENOENT	A message-queue identifier does not exist for the union of *key* and sensitivity label; 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 systemwide would be exceeded.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

Sensitivity labels are used together with *key* to determine message-queue identifiers.

SEE ALSO

intro(2TSOL), **msgctl**(2TSOL), **msgop**(2TSOL), **stdipc**(3C)

NAME

msgop, msgsnd, msgsndl, msgrcv, msgrcvl – Message operations

SYNOPSIS

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

int msgsnd(int *msqid*, **const void** ∗*msgp*, **size_t** *msgsz*, **int** *msgflg***);**

int msgrcv(int *msqid*, **void** ∗*msgp*, **size_t** *msgsz*, **long** *msgtyp*, **int** *msgflg***);**

cc [*flag* **...]** *file* **... -ltsol [** *library* **...]**
#include <sys/tsol/ipcl.h>

int msgsndl(int *msqid*, **const void** ∗*msgp*, **size_t** *msgsz*, **int** *msgflg*,
        const bilabel_t ∗*ilabel***);**
int msgrcvl(int *msqid*, **void** ∗*msgp*, **size_t** *msgsz*, **long** *msgtyp*, **int** *msgflg*,
        bilabel_t ∗*ilabel***);**

DESCRIPTION

In Trusted Solaris, individual messages within a message queue are labeled with an information label. This label does not float and cannot be modified once the message has been placed in the queue. The message-queue sensitivity label must dominate the information labels of all messages within the queue.

msgsnd( ) sends a message to the queue associated with the message-queue identifier specified by *msqid*. *msgp* points to a user-defined buffer that must contain first a field of type long integer that will specify the type of the message, and then a data portion that will hold the text of the message.

msgsnd( ) uses the current information label of the process to label a message. **msgsndl**( ) is identical to **msgsnd** but allows the calling process to label a message with a particular information label as specified by *ilabel*. If *ilabel* does not match the current information label of the process, **msgsndl** requires that the effective privilege set of the calling process include **PRIV_IPC_UPGRADE_IL** if the supplied label specifies an upgrade or **PRIV_IPC_DOWNGRADE_IL** if the supplied label specifies a downgrade.

Both **msgsnd** and **msgsndl** require either that a process have discretionary and mandatory write access to *msqid*, or that the effective privilege set of the calling process include **PRIV_IPC_DAC_WRITE** and **PRIV_IPC_MAC_WRITE**.

If information labels are enabled on this system and **PRIV_IPC_NOFLOAT** is asserted, no information label is placed on the message. Therefore the receiving process will not float as a result of receiving this message.

The following is an example of members that might be in a user-defined buffer (**mymsg**).

```
long mtype;        /* message type */
char mtext[];      /* message text */
```

mtype is a positive integer that can be used by the receiving process for message selection. **mtext** is any text of length *msgsz* bytes. *msgsz* can range from **0** to a system-imposed maximum.

msgflg specifies the action to be taken if one or both of these conditions are true:

The number of bytes already on the queue is equal to **msg_qbytes** [see **intro**(2TSOL)].

The total number of messages on all queues systemwide is equal to the system-imposed limit.

These are the actions to take:

If (*msgflg*&**IPC_NOWAIT**) is true, the message is not sent and the calling process returns immediately.

If (*msgflg*&**IPC_NOWAIT**) is false, the calling process suspends execution until one of these conditions occurs:

- The condition responsible for the suspension no longer exists, in which case the message is sent.
- *msqid* is removed from the system. [See **msgctl**(2TSOL).] When this removal occurs, **errno** is set to **EIDRM**, and a value of −**1** is returned.
- The calling process receives a signal that is to be caught. In this case, the message is not sent and the calling process resumes execution in the manner prescribed in **signal**(3C).

msgrcv( ) reads a message from the queue associated with the message-queue identifier specified by *msqid* and places the message in the user-defined structure to which *msgp* points. The structure must contain a message-type field followed by the area for the message text. (See the structure **mymsg** earlier.) **mtype** is the received message's type as specified by the sending process. **mtext** is the text of the message. *msgsz* specifies the size in bytes of **mtext**.

msgrcvl( ) is identical to **msgrcv**( ) but allows the calling process to retrieve the information label *ilabel*associated with a message.

Both **msgrcvl**( ) and **msgrcv**( ) require either that a process have discretionary and mandatory read access to *msqid*, or that the effective privilege set of the calling process include one or both of these privileges to override the corresponding check: **PRIV_IPC_DAC_READ** and **PRIV_IPC_MAC_READ** privileges. In addition, both calls must either have mandatory write access to *msqid* or have **PRIV_IPC_MAC_WRITE** asserted.

The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg*&**MSG_NOERROR**) is true. The truncated part of the message is lost and no indication of the truncation is given to the calling process.

msgtyp specifies the type of message requested:

If *msgtyp* is 0, the first message on the queue is received.

If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.

If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

msgflg specifies the action to be taken if a message of the desired type is not on the queues:

If (*msgflg***&IPC_NOWAIT**) is true, the calling process returns immediately with a return value of −**1** and sets **errno** to **ENOMSG**.

If (*msgflg***&IPC_NOWAIT**) is false, the calling process suspends execution until one of these conditions occurs:

- A message of the desired type is placed on the queue.
- *msqid* is removed from the system. When this removal occurs, **errno** is set to **EIDRM**, and a value of −**1** is returned.
- The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes execution in the manner prescribed in **signal**(3C).

If information-label floating is enabled on this system, retrieving a message from a queue using **msgrcv** or **msgrcvl** can cause the information label of the calling process to float up to the information label of the message. The float will fail if the resulting information label is not dominated by the sensitivity label of the process; this failure will in turn cause **msgrcv** or **msgrcvl** to fail unless the effective privilege set of the calling process includes **PRIV_PROC_NOFLOAT**. If this failure occurs, the message is lost.

RETURN VALUES

If **msgsnd**( ), **msgsndl**( ), **msgrcv**( ), or **msgrcvl**( ), return because of the receipt of a signal, they return a value of −**1** and set **errno** to **EINTR**. If they return because of removal of *msqid* from the system, they return a value of −**1** and set **errno** to **EIDRM**.

Upon successful completion, **msgsnd** returns a value of **0** and **msgrcv** returns the number of bytes actually placed into *mtext*.

Upon failure, they return a value of −**1** and set **errno** to indicate the error.

ERRORS

msgsnd and **msgsndl** fail and send no message if any of these conditions is true:

EACCES Operation permission is denied to the calling process [see **intro**(2TSOL)], and the process did not have the appropriate privilege in its set of effective privileges.

The sensitivity label of *msqid* does not match the sensitivity label of the calling process, and the calling process does not have the appropriate privilege override(s) in its set of effective privileges.

EAGAIN The message cannot be sent for one of the reasons cited earlier and (*msgflg***&IPC_NOWAIT**) is true.

EFAULT *msgp* points to an illegal address.

ilabel points to an illegal address.

EINVAL *msqid* is not a valid message-queue identifier.

mtype is less than 1.

msgsz is less than zero or greater than the system-imposed limit.

EINVAL The label to which *ilabel* points is not a valid information label.

If sent to the message queue, this message would cause the information label of the message to dominate the sensitivity label of the queue, and the calling process does not have the appropriate privilege override(s) in its set of effective privileges.

EPERM This call is trying either to upgrade or to downgrade the information label but is not suitably privileged.

Upon successful completion, these actions are taken with respect to the data structure associated with *msqid* [see **intro**(2TSOL)]:

msg_qnum is incremented by 1.

msg_lspid is set to the process ID of the calling process.

msg_stime is set to the current time.

msgrcv( ) and **msgrcvl**( ) fail and receive no message if any of these conditions is true:

E2BIG The length of *mtext* is greater than *msgsz* and (*msgflg*&**MSG_NOERROR**) is false.

EACCES Operation permission is denied to the calling process, and the calling process does not have the appropriate privilege override(s) in its set of effective privileges.

The sensitivity label of *msqid* does not match the sensitivity label of the calling process, and the calling process does not have the appropriate privilege override(s) in its set of effective privileges.

EFAULT *msgp* points to an illegal address.

ilabel points to an illegal address.

EINVAL *msqid* is not a valid message-queue identifier.

msgsz is less than 0.

The read from the message queue would cause the information label of the calling process to float above its sensitivity label, and the calling process does not have the **PRIV_PROC_NOFLOAT** privilege in its set of effective privileges.

ENOMSG The queue does not contain a message of the desired type and (*msgtyp*&**IPC_NOWAIT**) is true.

Upon successful completion, these actions are taken with respect to the data structure associated with **msqid** [see **intro**(2TSOL)]:

msg_qnum is decremented by 1.

msg_lrpid is set to the process ID of the calling process.

msg_rtime is set to the current time.

SUMMARY OF TRUSTED SOLARIS

CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

SEE ALSO

intro(2TSOL), **msgctl**(2TSOL), **msgget**(2TSOL), **signal**(3C)

NAME

nice – Change priority of a process

SYNOPSIS

#include <unistd.h>

int nice(int *incr***);**

DESCRIPTION

nice( ) allows a process to change its priority. The invoking process must be in a scheduling class that supports the **nice** system call. The **priocntl** function is a more general interface to scheduler functions.

nice adds the value of *incr* to the nice value of the calling process. A nice value of a process is a nonnegative number for which a more positive value results in lower CPU priority.

A maximum nice value of 39 and a minimum nice value of 0 are imposed by the system. (The default nice value is 20.) Requests for values above or below these limits result in the nice value being set to the corresponding limit.

RETURN VALUES

Upon successful completion, **nice** returns the new nice value minus 20. Upon failure, **nice** returns a value of –**1** and sets **errno** to indicate the error.

ERRORS

nice fails if either of these conditions is true:

EINVAL	**nice** is called by a process in a scheduling class other than time-sharing.
EPERM	*incr* is negative or greater than 40 and the **PRIV_SYS_CONFIG** privilege of the calling process is not asserted.

SUMMARY OF TRUSTED SOLARIS CHANGES

Use of the **PRIV_SYS_CONFIG** privilege replaces the super-user check in base Solaris.

SEE ALSO

nice(1TSOL), **exec**(2TSOL), **priocntl**(2TSOL)

NAME

open – Open for reading or writing

SYNOPSIS

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

int open(const char **path*, **int** *oflag*, **/* mode_t** *mode* ***/ ...);**

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

open( ) opens a file descriptor for the file to which *path* points and sets the file status flags according to the value of *oflag*. *oflag* values are constructed by OR-ing flags from the following list:

NOTE: The first three flags are mutually exclusive.

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY or **O_NONBLOCK**
These flags may affect subsequent reads and writes. [See **read**(2TSOL) and **write**(2TSOL).] 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_NDELAY** or **O_NONBLOCK** is set, an **open** for reading only will return without delay; an **open** for writing only will return an error if no process currently has the file open for reading.

If **O_NDELAY** and **O_NONBLOCK** are clear, an **open** for reading only will block until a process opens the file for writing; an **open** for writing only will block until a process opens the file for reading.

When opening a file associated with a terminal line:

If **O_NDELAY** or **O_NONBLOCK** is set, the **open** will return without waiting for the device to be ready or available; subsequent behavior of the device is device specific.

If **O_NDELAY** and **O_NONBLOCK** are clear, the **open** will block until the device is ready or available.

O_APPEND If set, the file pointer will be set to the end of the file prior to each write.

O_DSYNC Write I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion.

O_RSYNC Read I/O operations on the file descriptor complete at the same level of integrity as specified by the **O_DSYNC** and **O_SYNC** flags. If both

O_DSYNC and **O_RSYNC** are set in *oflag*, all I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion. If both **O_SYNC** and **O_RSYNC** are set in *oflag*, all I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.

O_SYNC When opening a regular file, this flag affects subsequent writes. If set, each **write**(2TSOL) will wait for both the file data and file status to be physically updated. Write I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.

O_NOCTTY If set and the file is a terminal, the terminal will not be allocated as the controlling terminal of the calling process.

O_CREAT If the file exists, this flag has no effect except as noted under **O_EXCL**. Otherwise, the file is created and the owner ID of the file is set to the effective user ID of the process; the group ID of the file is set to the effective group ID of the process; or if the S_ISGID bit is set in the directory in which the file is being created, the file's group ID is set to the group ID of its parent directory. If the group ID of the new file does not match the effective group ID or one of the supplementary groups IDs, the **S_ISGID** bit is cleared. The calling process must assert the **PRIV_FILE_SETID** privilege to override clearing the S_ISGID bit. The access permission bits of the file mode are set to the value of *mode*, modified as follows: [See **creat**(2TSOL).]

- All bits set in the file mode-creation mask of the process are cleared. [See **umask**(2).]
- The "save text image after execution bit" of the mode is cleared. [See **chmod**(2TSOL).] **O_SYNC** write I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion. [See **fcntl**(5) definition of **O_SYNC .]**
- The calling process must assert the **PRIV_SYS_CONFIG** privilege to override clearing the S_ISVTX bit.

O_EXCL If **O_EXCL** and **O_CREAT** are set, **open** will fail if the file exists. The check for the existence of the file and the creation of the file if it does not exist are atomic with respect to other processes executing **open** naming the same file name in the same directory with **O_EXCL** and **O_CREAT** set.

O_TRUNC If the file exists, its length is truncated to 0; and the mode and owner are unchanged. **O_TRUNC** has no effect on FIFO special files or directories.

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

When **open** is invoked to open a named stream, and the **connld** module [see **connld**(7M)] has been pushed on the pipe, **open** blocks until the server process has issued an **I_RECVFD ioctl** [see **streamio**(7I)] to receive the file descriptor.

If *path* is a symbolic link and **O_CREAT** and **O_EXCL** are set, the link is not followed.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is the lowest-numbered file descriptor available and is set to remain open across **exec** functions. [See **fcntl**(2).]

As described in **fcntl**(2), certain flag values can be set following **open**.

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.

For file system objects that support exclusive (**OEXCL**) open (such as *procfs*) or exclusive (**TIOCEXCL**) access (such as many devices), the calling process may assert the **PRIV_SYS_DEVICES** privilege to override the exclusive restriction.

In the case of *procfs*, the calling process cannot open a process whose program file has the set-user-ID or set-group-ID mode bits set, or has the use of privilege. The calling process may assert the **PRIV_PROC_OWNER** privilege to bypass this restriction.

Directories can be opened only for reading. There is no privilege to override this restriction.

The information label of the calling process is unchanged. However, when the **open** causes the specified file to be truncated, the information label of the file is set to ADMIN_LOW. Otherwise, the file information label is unchanged.

RETURN VALUES

Upon successful completion, **open** returns the file descriptor. Upon failure, **open** returns –**1** and sets **errno** to indicate the error.

ERRORS

The named file is opened unless any of these conditions is true:

EACCES The file does not exist and write permission is denied by the parent directory of the file to be created. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_WRITE** and **PRIV_FILE_MAC_WRITE**.

EACCES **O_TRUNC** is specified and write permission is denied To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_WRITE** and **PRIV_FILE_MAC_WRITE**.

A component of the path prefix denies search permission. To

	override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_SEARCH** and **PRIV_FILE_MAC_SEARCH**.
	O_RDWR or **O_WRONLY** is specified and the calling process does not have write access to the file. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_WRITE** and **PRIV_FILE_MAC_WRITE**.
	O_RDWR or **O_RDONLY** is specified, the file is opened for reading, and the calling process does not have read access to the file. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_READ** and **PRIV_FILE_MAC_READ**.
EAGAIN	The file exists with enforced record locking enabled, record locks are on the file [see **chmod**(2)], and **O_TRUNC** is specified.
EDQUOT	The file does not exist, **O_CREAT** is specified, and the directory in which 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 file does not exist, **O_CREAT** is specified, and the user's quota of inodes on the file system in which the file is being created has been exhausted.
EEXIST	**O_CREAT** and **O_EXCL** are set, and the named file exists.
EFAULT	*path* points to an illegal address.
EINTR	A signal was caught during the **open** function.
EIO	A hangup or error occurred during the open of the STREAMS-based device.
EISDIR	The named file is a directory and *oflag* is write or read / write.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMFILE	The process has too many open files. [See **getrlimit**(2TSOL).]
EMULTIHOP	Components of *path* require hopping to multiple remote machines and the file system does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENFILE	The system file table is full.
ENOENT	**O_CREAT** is not set and the named file does not exist.
	O_CREAT is set and a component of the path prefix does not exist or is a null path name.
ENOLINK	*path* points to a remote machine, but the link to that machine is no longer active.

ENOMEM	The system is unable to allocate a send descriptor.
ENOSPC	**O_CREAT** and **O_EXCL** are set and the file system is out of inodes.
	O_CREAT is set and the directory that would contain the file cannot be extended.
ENOSR	Unable to allocate a stream
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The named file is a character special or block special file, and the device associated with this special file does not exist.
	O_NDELAY or **O_NONBLOCK** is set, the named file is a FIFO, **O_WRONLY** is set, and no process has the file open for reading.
	A STREAMS module or driver open routine failed.
EROFS	The named file resides on a read-only file system and either **O_WRONLY, O_RDWR, O_CREAT,** or **O_TRUNC** is set in *oflag* (if the file does not exist).

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

To open a file system object that supports exclusive open or exclusive access, the calling process may assert the **PRIV_SYS_DEVICES** privilege. In the case of *procfs*, the calling process cannot open a process whose program file has the S_ISUID or S_ISGUID mode bits set or has the use of privilege. The calling process may assert the **PRIV_PROC_OWNER** privilege. When used to create a new file, the calling process may need to assert one or both of these privileges: **PRIV_SYS_CONFIG** to override clearing the S_ISVTX bit, and **PRIV_FILE_SETID** to override clearing the S_ISGID bit.

SEE ALSO

intro(2TSOL), **chmod**(2TSOL), **close**(2), **creat**(2TSOL), **dup**(2), **exec**(2TSOL), **fcntl**(2TSOL), **getmsg**(2), **getrlimit**(2TSOL), **lseek**(2TSOL), **putmsg**(2), **read**(2TSOL), **stat**(2TSOL), **umask**(2), **write**(2TSOL), **fcntl**(5), **stat**(5), **connld**(7M), **streamio**(7I)

NAME

p_online – change processor online or offline status

SYNOPSIS

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

int p_online(processorid_t *processorid* **, int** *flag* **);**

DESCRIPTION

The processor specified by the first argument is set online or offline or is unchanged, depending on whether the *flag* argument is **P_ONLINE**, **P_OFFLINE**, or **P_STATUS**.

When a flag of **P_ONLINE** is specified, the processor, if previously offline, is brought online and allowed to process LWPs and perform system activities.

When **P_OFFLINE** is specified, and the processor is not already offline, it is taken offline and not allowed to process LWPs. The processor will become as inactive as possible.

When **P_STATUS** is specified, no change occurs, but the current status is returned.

RETURN VALUES

On successful completion, the value returned is the previous state of the processor, **P_ONLINE** or **P_OFFLINE**. Otherwise, a value of –1 is returned and errno is set to indicate the error.

ERRORS

EPERM	The calling process does not have the **PRIV_SYS_CONFIG** privilege.
EINVAL	An non-existent processor ID was specified or *flag* was invalid.
EBUSY	*flag* was **P_OFFLINE** and the specified processor is the only online processor, there are currently LWPs bound to the processor, or the processor performs some essential function that cannot be performed by another processor.

SUMMARY OF TRUSTED SOLARIS CHANGES

The calling process must have the **PRIV_SYS_CONFIG** privilege in order to perform the P_ONLINE and P_OFFLINE operations.

SEE ALSO

psradm(1M), **psrinfo**(1M), **processor_bind**(2TSOL), **processor_info**(2), **sysconf**(3C)

NAME

priocntl – Control process schedulers

SYNOPSIS

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

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

DESCRIPTION

priocntl( ) provides for control over the scheduling of an active light-weight process (LWP).

LWPs fall into distinct classes with a separate scheduling policy applied to each class. The two classes currently supported are the real-time class and the time-sharing class. The characteristics of these classes are described under the corresponding headings that follow. The class attribute of an LWP is inherited across the **fork**(2TSOL), **exec**(2TSOL)**,** and **_lwp_create**(2) system calls. **priocntl** can be used to dynamically change the class and other scheduling parameters associated with a running LWP or set of LWPs given the appropriate permissions explained hereafter.

In the default configuration, a runnable real-time LWP runs before any other LWP. Therefore, inappropriate use of a real-time LWP can have a dramatic negative impact on system performance.

priocntl provides an interface for specifying a process, set of processes, or an LWP to which the function is to apply. The **priocntlset**(2TSOL) system call provides the same functions as **priocntl** but allows a more general interface for specifying the set of LWPs to which the function is to apply.

For **priocntl**, the *idtype* and *id* arguments are used together to specify the set of LWPs. The interpretation of *id* depends on the value of *idtype*. These are possible values for *idtype* and corresponding interpretations of *id*:

P_LWPID	*id* is an LWP ID. The **priocntl**( ) system call applies to the LWP with the specified ID within the calling process.
P_PID	*id* is a process ID specifying a single process. The **priocntl** system call applies to all LWPs currently associated with the specified process.
P_PPID	*id* is a parent process ID. The **priocntl** system call applies to all LWPs currently associated with processes with the specified parent process ID.
P_PGID	*id* is a process group ID. The **priocntl** system call applies to all LWPs currently associated with processes in the specified process group.
P_SID	*id* is a session ID. The **priocntl** system call applies to all LWPs currently associated with processes in the specified session.
P_CID	*id* is a class ID (returned by **priocntl PC_GETCID** as explained hereafter). The **priocntl** system call applies to all LWPs in the specified class.
P_UID	*id* is a user ID. The **priocntl** system call applies to all LWPs with this effective user ID.

P_GID *id* is a group ID. The **priocntl** system call applies to all LWPs with this effective group ID.

P_ALL The **priocntl** system call applies to all existing LWPs. The value of *id* is ignored. The permission restrictions described below still apply.

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

In order to change the scheduling parameters of an LWP (using the **PC_SETPARMS** command as explained hereafter) the calling LWP must have process MAC write access, and 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 calling LWP must have the **PRIV_PROC_OWNER** privilege. These are the minimum permission requirements enforced for all classes. An individual class may impose additional permissions requirements when setting LWPs to that class and / or when setting class-specific scheduling parameters.

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

The **init** process is a special case. In order for a **priocntl** call to change the class or other scheduling parameters of the **init** process (process ID 1), it must be the only process specified by *idtype* and *id*. The **init** process may be assigned to any class configured on the system, but the time-sharing class is almost always the appropriate choice. (Other choices may be highly undesirable; see the *UNKNOWN TITLE ABBREVIATION: SYSADMIN2* for more information.)

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

A structure with the following members is used by the **PC_GETCID** and **PC_GETCLINFO** commands:

```
id_t    pc_cid;                     /* Class id */
char    pc_clname[PC_CLNMSZ];       /* Class name */
long    pc_clinfo[PC_CLINFOSZ];     /* Class information */
```

pc_cid is a class ID returned by **priocntl PC_GETCID**. **pc_clname** is a buffer of size **PC_CLNMSZ** (defined in **<sys/priocntl.h>**) used to hold the class name (**RT** for real-time or **TS** for time-sharing).

pc_clinfo is a buffer of size **PC_CLINFOSZ** (defined in **<sys/priocntl.h>**) used to return data describing the attributes of a specific class. The format of this data is class-specific and is described under the appropriate heading (**REAL-TIME CLASS** or **TIME-SHARING CLASS**) following.

A structure with the following elements is used by the **PC_SETPARMS** and **PC_GETPARMS** commands:

```
id_t    pc_cid;                        /* LWP class */
long    pc_clparms[PC_CLPARMSZ];       /* Class-specific params */
```

pc_cid is a class ID (returned by **priocntl PC_GETCID**). The special class ID **PC_CLNULL** can also be assigned to **pc_cid** when using the **PC_GETPARMS** command as explained hereafter.

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. **PC_CLPARMSZ** is the length of the **pc_clparms** buffer and is defined in **<sys/priocntl.h>**.

Commands

These are available **priocntl** commands:

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 not 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 (including the **sys** class) configured in the system. If the class specified by **pc_clname** is invalid or is not currently configured, the **priocntl** call returns **–1** with **errno** set to **EINVAL**. The format of the attribute data returned for a given class is defined in the **<sys/rtpriocntl.h>** or **<sys/tspriocntl.h>** header file and described under the appropriate heading hereafter.

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

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

On success, the class name is returned in the **pc_clname** buffer, the class attributes are returned in the **pc_clinfo** buffer, and the **priocntl** call returns the total number of classes (including the **sys** class) configured in the system. 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 hereafter.

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 it is used with the *idtype* of **P_LWPID**, this command will set the class and class-specific scheduling parameters of the LWP. *arg* points to a structure of type **pcparms_t**. **pc_cid** specifies the class you are setting, and the **pc_clparms** buffer contains the class-specific parameters you are setting. The format of the class-specific parameter data is defined in the **<sys/rtpriocntl.h>** or **<sys/tspriocntl.h>** header and described under the appropriate class heading hereafter.

When setting parameters for a set of LWPs, **priocntl** acts on the LWPs in the set in an implementation-specific order. If it encounters an error for one or more of the target processes, **priocntl** may or may not continue through the set of LWPs depending on the nature of the error. If the error is related to permissions (**EPERM**), **priocntl** continues through the LWP set, resetting the parameters for all target LWPs for which the calling LWP has appropriate permissions. **priocntl** then returns −**1** with **errno** set to **EPERM** to indicate that the operation failed for one or more of the target LWPs. If it encounters an error other than permissions, **priocntl** does not continue through the set of target LWPs but returns the error immediately.

PC_GETPARMS Get the class and / or class-specific scheduling parameters of an LWP. *arg* points to a structure of type **pcparms_t**.

If **pc_cid** specifies a configured class and a single LWP belonging to that class is specified by the *idtype* and *id* values or the **procset** structure, then the scheduling parameters of that LWP are returned in the **pc_clparms** buffer. If the LWP specified does not exist or does not belong to the specified class, the **priocntl** call returns −**1** with **errno** set to **ESRCH**.

If **pc_cid** specifies a configured class and a set of LWPs is specified, the scheduling parameters of one of the specified LWPs 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 exists or none of them belongs 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**(1MTSOL) command and is not intended for general use by other applications.

REAL-TIME CLASS

The real-time class provides a fixed-priority, preemptive scheduling policy for those LWPs requiring fast and deterministic response and absolute user / application control of scheduling priorities. If it is configured in the system, the real-time class should have exclusive control of the highest range of scheduling priorities on the system to ensure that a runnable real-time LWP is given CPU service before any LWP belonging to any other class.

The real-time class has a range of real-time priority (**rt_pri**) values that may be assigned to an LWP within the class. Real-time priorities range from 0 to *x* where the value of *x* is configurable and can be determined for a specific installation by using the **priocntl PC_GETCID** or **PC_GETCLINFO** command.

The real-time scheduling policy is a fixed-priority policy. The scheduling priority of a real-time LWP is never changed except as the result of an explicit request by the user / application to change the **rt_pri** value of the LWP.

For an LWP in the real-time class, the **rt_pri** value is, for all practical purposes, equivalent to the scheduling priority of the LWP. The **rt_pri** value completely determines the scheduling priority of a real-time LWP relative to other LWPs within its class. Numerically higher **rt_pri** values represent higher priorities. Since it controls the highest range of scheduling priorities in the system, the real-time class is guaranteed that the runnable real-time LWP with the highest **rt_pri** value is always selected to run before any other LWPs in the system.

In addition to providing control over priority, **priocntl** provides for control over the length of the time quantum allotted to the LWP in the real-time class. The time quantum value specifies the maximum amount of time an LWP may run assuming that it does not complete or enter a resource or event wait state (**sleep**). Note that if another LWP becomes runnable at a higher priority, the currently running LWP may be preempted before receiving its full time quantum.

The system's process scheduler keeps the runnable real-time LWPs on a set of scheduling queues. There is a separate queue for each configured real-time priority, and all real-time LWPs with a given **rt_pri** value are kept together on the appropriate queue. The LWPs on a given queue are ordered in FIFO order (that is, the LWP at the front of the queue has been waiting longest for service and receives the CPU first). Real-time LWPs that wake up after sleeping, LWPs that change to the real-time class from some other class, LWPs that have used their full time quantum, and runnable LWPs whose priority is reset by **priocntl** are all placed at the back of the appropriate queue for their priority. An LWP that is preempted by a higher priority LWP remains at the front of the queue (with whatever time is remaining in its time quantum) and runs before any other LWP at this priority. Following **fork**(2TSOL) or **_lwp_create**(2) system calls by a real-time LWP, the parent LWP continues to run while the child LWP (which inherits its parent's **rt_pri** value) is placed at the back of the queue.

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

```
short      rt_maxpri;      /* Maximum real-time priority */
```

The **priocntl PC_GETCID** and **PC_GETCLINFO** commands return real-time class attributes in the **pc_clinfo** buffer in this format.

rt_maxpri specifies the configured maximum **rt_pri** value for the real-time class. (If **rt_maxpri** is *x*, the valid real-time priorities range from 0 to *x*.)

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

```
short    rt_pri;        /* Real-Time priority */
ulong    rt_tqsecs;     /* Seconds in time quantum */
long     rt_tqnsecs;    /* Additional nanoseconds in quantum */
```

When using the **priocntl PC_SETPARMS** or **PC_GETPARMS** commands, if **pc_cid** specifies the real-time class, the data in the **pc_clparms** buffer is in this format.

The aforementioned commands can be used to set the real-time priority to the specified value or get the current **rt_pri** value. Setting the **rt_pri** value of an LWP that is currently running or runnable (not sleeping) causes the LWP to be placed at the back of the scheduling queue for the specified priority. The LWP is placed at the back of the appropriate queue regardless of whether the priority being set is different from the previous **rt_pri** value of the LWP. Note that a running LWP can voluntarily release the CPU and go to the back of the scheduling queue at the same priority by resetting its **rt_pri** value to its current real-time priority value. In order to change the time quantum of an LWP without setting the priority or affecting the LWP's position on the queue, the **rt_pri** field should be set to the special value **RT_NOCHANGE** (defined in **<sys/rtpriocntl.h>**). Specifying **RT_NOCHANGE** when changing the class of an LWP to real-time from some other class results in the real-time priority being set to zero.

For the **priocntl PC_GETPARMS** command, if **pc_cid** specifies the real-time class and more than one real-time LWP is specified, the scheduling parameters of the real-time LWP with the highest **rt_pri** value among the specified LWPs are returned and the LWP ID of this LWP is returned by the **priocntl** call. If there is more than one LWP sharing the highest priority, the one returned is implementation- dependent.

The **rt_tqsecs** and **rt_tqnsecs** fields are used for getting or setting the time quantum associated with an LWP or group of LWPs. **rt_tqsecs** is the number of seconds in the time quantum and **rt_tqnsecs** is the number of additional nanoseconds in the quantum. For example setting **rt_tqsecs** to 2 and **rt_tqnsecs** to 500,000,000 (decimal) would result in a time quantum of two and one-half seconds. Specifying a value of 1,000,000,000 or greater in the **rt_tqnsecs** field results in an error return with **errno** set to **EINVAL**. Although the resolution of the **tq_nsecs** field is very fine, the specified time quantum length is rounded up by the system to the next integral multiple of the system clock's resolution. The maximum time quantum that can be specified is implementation-specific and equal to **LONG_MAX** ticks (defined in **<limits.h>**). Requesting a quantum greater than this maximum results in an error return with **errno** set to **ERANGE** (although infinite quantums may be requested using a special value as explained hereafter). Requesting a time quantum of zero (setting both **rt_tqsecs** and **rt_tqnsecs** to 0) results in an error return with **errno** set to **EINVAL**.

The **rt_tqnsecs** field can also be set to one of the following special values (defined in **<sys/rtpriocntl.h>**), in which case the value of **rt_tqsecs** is ignored:

RT_TQINF	Set an infinite time quantum.
RT_TQDEF	Set the time quantum to the default for this priority. [See **rt_dptbl**(4).]
RT_NOCHANGE	Do not set the time quantum. This value is useful when you wish to change the real-time priority of an LWP without affecting the time quantum. Specifying this value when changing the class of an LWP to real-time from some other class is equivalent to specifying **RT_TQDEF**.

In order to change the class of an LWP to real-time (from any other class), the LWP invoking **priocntl** must have the **PRIV_SYS_CONFIG** privilege. In order to change the priority or time quantum setting of a real-time LWP, the LWP invoking **priocntl** must have the **PRIV_PROC_OWNER** privilege or must itself be a real-time LWP whose real or effective user ID matches the real of effective user ID of the target LWP.

The real-time priority and time quantum are inherited across the **fork**(2TSOL) and **exec**(2TSOL) system calls.

TIME-SHARING CLASS

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

The time-sharing class has a range of time-sharing user priority (see **ts_upri** hereafter) values that may be assigned to LWPs within the class. A **ts_upri** value of zero is defined as the default base priority for the time-sharing class. User priorities range from $-x$ to $+x$ where the value of x is configurable and can be determined for a specific installation by using the **priocntl PC_GETCID** or **PC_GETCLINFO** command.

The purpose of the user priority is to provide some degree of user / application control over the scheduling of LWPs in the time-sharing class. Raising or lowering the **ts_upri** value of an LWP in the time-sharing class raises or lowers the scheduling priority of the LWP. It is not guaranteed, however, that an LWP with a higher **ts_upri** value will run before one with a lower **ts_upri** value 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 systemwide limits on user priority (returned by the **PC_GETCID** and **PC_GETCLINFO** commands) there is a per LWP user priority limit (see **ts_uprilim** hereafter), which specifies the maximum **ts_upri** value that may be set for a given LWP; by default, **ts_uprilim** is zero.

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

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

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

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

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

```
short    ts_uprilim;    /* Time-Sharing user priority limit */
short    ts_upri;       /* Time-Sharing user priority */
```

When using the **priocntl PC_SETPARMS** or **PC_GETPARMS** commands, if **pc_cid** specifies the time-sharing class, the data in the **pc_clparms** buffer is in this format.

For the **priocntl PC_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 the **PRIV_SYS_CONFIG** privilege may raise a **ts_uprilim**. When changing the class of an LWP to time-sharing from some other class, the **PRIV_SYS_CONFIG** privilege is required in order to set the initial **ts_uprilim** to a value greater than zero. Attempts by a nonprivileged LWP to raise a **ts_uprilim** or set an initial **ts_uprilim** greater than zero fail with a return value of –**1** and **errno** set to **EPERM**.

Any time-sharing LWP may set its own **ts_upri** (or that of another LWP with the same user ID) to any value less than or equal to the LWP's **ts_uprilim**. Attempts to set the **ts_upri** above the **ts_uprilim** (and / or set the **ts_uprilim** below the **ts_upri**) result in the **ts_upri** being set equal to the **ts_uprilim**.

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

The time-sharing user priority and user-priority limit are inherited across the **fork** and **exec** functions.

RETURN VALUES

Unless otherwise noted earlier, **priocntl** returns a value of **0** on success. **priocntl** returns –**1** on failure and sets **errno** to indicate the error.

ERRORS

priocntl fails if any of these conditions is true:

EAGAIN	An attempt to change the class of an LWP failed because of insufficient resources other than memory (for example, class-specific kernel data structures).
EFAULT	One of the arguments points to an illegal address.
EINVAL	The argument *cmd* was invalid, an invalid or unconfigured class was specified, or one of the parameters specified was invalid.
ENOMEM	An attempt to change the class of an LWP failed because of insufficient memory.
EPERM	The calling LWP does not have required privileges.
ERANGE	The requested time quantum is out of range.
ESRCH	None of the specified LWPs exists.

SUMMARY OF TRUSTED SOLARIS CHANGES

Super-user checks in base Solaris are replaced by privilege checks. MAC policy is enforced in addition to DAC.

SEE ALSO

priocntl(1), **dispadmin**(1MTSOL), **init**(1M), **_lwp_create**(2), **exec**(2TSOL), **fork**(2TSOL), **nice**(2TSOL), **priocntlset**(2TSOL), **rt_dptbl**(4)

System Interface Guide

NAME

priocntlset – Control process scheduler

SYNOPSIS

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

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

DESCRIPTION

priocntlset( ) changes the scheduling properties of running processes. **priocntlset** has the same functions as the **priocntl**( ) function but a more general way of specifying the set of processes whose scheduling properties are to be changed.

cmd specifies the function to be performed. *arg* is a pointer to a structure whose type depends on *cmd*. See **priocntl**(2TSOL) for the valid values of *cmd* and the corresponding *arg* structures.

psp is a pointer to a **procset** structure, which **priocntlset** uses to specify the set of processes whose scheduling properties are to be changed. The **procset** structure contains the following members:

```
idop_t      p_op;       /* operator connecting left/right sets */
idtype_t    p_lidtype;  /* left set ID type */
id_t        p_lid;      /* left set ID */
idtype_t    p_ridtype;  /* right set ID type */
id_t        p_rid;      /* right set ID */
```

p_lidtype and **p_lid** specify the ID type and ID of one (left) set of processes; **p_ridtype** and **p_rid** specify the ID type and ID of a second (right) set of processes. ID types and IDs are specified just as for the **priocntl**( ) function. **p_op** specifies the operation to be performed on the two sets of processes to get the set of processes the function is to apply to. The valid values for **p_op** and the processes they specify are

POP_DIFF	Set difference: processes in left set and not in right set
POP_AND	Set intersection: processes in both left and right sets
POP_OR	Set union: processes in either left or right sets or both
POP_XOR	Set EXCLUSIVE OR: processes in left or right set but not in both

The following macro, which is defined in **procset.h**, offers a convenient way to initialize a **procset** structure:

```
#define setprocset(psp, op, ltype, lid, rtype, rid) \
(psp)→p_op       = (op), \
(psp)→p_lidtype  = (ltype), \
(psp)→p_lid      = (lid), \
(psp)→p_ridtype  = (rtype), \
(psp)→p_rid      = (rid),
```

RETURN VALUES

Upon success, **priocntlset** returns a value of **0**. **priocntlset** returns –**1** on failure and sets **errno** to indicate the error.

ERRORS

priocntlset fails if any of these conditions is true:

EAGAIN	An attempt to change the class of a process failed because of insufficient resources other than memory (for example, class-specific kernel data structures).
EFAULT	One of the arguments points to an illegal address.
EINVAL	The argument *cmd* was invalid, an invalid or unconfigured class was specified, or one of the parameters specified was invalid.
ENOMEM	An attempt to change the class of a process failed because of insufficient memory.
EPERM	The calling process does not have required privileges.
ERANGE	The requested time quantum is out of range.
ESRCH	None of the specified processes exists.

SUMMARY OF TRUSTED SOLARIS CHANGES

This system call is a generalized interface of the **priocntl**(2TSOL) system call. Privilege requirements as described in **priocntl**(2TSOL) apply here as well.

SEE ALSO

priocntl(1), **priocntl**(2TSOL)

NAME

processor_bind – bind LWPs to a processor

SYNOPSIS

#include <sys/types.h>
#include <sys/processor.h>
#include <sys/procset.h>

int processor_bind(idtype_t *idtype*, **id_t** *id*, **processorid_t** *processorid*,
processorid_t **obind***);**

DESCRIPTION

The LWP or set of LWPs specified by *idtype* and *id* are bound to the processor specified by *processorid*. Additionally, if *obind* is not **NULL**, the *processorid_t* variable pointed to by *obind* will be set to the previous binding of one of the specified LWPs, or to **PBIND_NONE** if the selected LWP was not bound.

If *idtype* is **P_PID**, the binding effects all LWPs of the process with process ID (PID) *id*.

If *idtype* is **P_LWPID**, the binding effects the LWP of the current process with LWP ID *id*.

If *id* is **P_MYID**, the specified LWP or process is the current one.

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

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

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

RETURN VALUES

processor_bind returns **0** if successful; otherwise, –**1** is returned and **errno** is set to reflect the error.

ERRORS

ESRCH	No processes or LWPs were found to match the criteria specified by *idtype* and *id*.
EINVAL	An non-existent or offline processor was specified.
EINVAL	*idtype* was not **P_PID** or **P_LWPID**.
EFAULT	The location pointed to by *obind* was not **NULL** and not writable by the user.
EPERM	The calling process does not have the **PRIV_PROC_OWNER** privilege, and its real or effective user ID does not match the real or effective user ID of one of the LWPs being bound.

SUMMARY OF TRUSTED SOLARIS CHANGES

The real or effective user ID of the calling process must match the real or effective user ID of the target process. The calling process must have the **PRIV_PROC_OWNER** privilege in order to override this restriction.

SEE ALSO **psradm**(1M), **psrinfo**(1M), **p_online**(2TSOL), **sysconf**(3C)

NAME

read, readl, pread, preadl, readv, readvl – Read from a file

SYNOPSIS

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

ssize_t read(int *fildes*, **void** ∗*buf*, **size_t** *nbyte***);**

ssize_t pread(int *fildes*, **void** ∗*buf*, **size_t** *nbyte*, **off_t** *offset***);**

ssize_t readv(int *fildes*, **struct iovec** ∗*iov*, **int** *iovcnt***);**

#include <tsol/rdwrl.h>
ssize_t readl(int *fildes*, **void** ∗*buf*, **size_t** *nbyte*, **bclabel_t** ∗*label_p***);**

ssize_t preadl(int *fildes*, **void** ∗*buf*, **size_t** *nbyte*, **off_t** *offset*, **bclabel_t** ∗*label_p***);**

ssize_t readvl(int *fildes*, **struct iovec** ∗*iov*, **int** *iovcnt*, **bclabel_t** ∗*label_p***);**

MT-LEVEL

read( ) is Async-Signal-Safe.

DESCRIPTION

read( ) attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer to which *buf* points. If *nbyte* is zero, **read** returns zero and has no other results. *fildes* is an open file descriptor.

On devices capable of seeking, the **read** starts at a position in the file given by the file pointer associated with *fildes*. On return from **read**, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

pread( ) performs the same action as **read**, except that **pread** reads from a given position in the file without changing the file pointer. The first three arguments for **pread** are the same as those for **read** with the addition of a fourth argument *offset* for the desired position inside the file. An attempt to perform a **pread** on a file that is incapable of seeking results in an error.

readv( ) performs the same action as **read**, but places the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], . . ., *iov*[*iovcnt*– 1].

The **iovec** structure contains these members:

```
caddr_t   iov_base;
int       iov_len;
```

Each **iovec** entry specifies the base address and length of an area in memory where data should be placed. **readv** always fills one buffer completely before proceeding to the next.

On success, **read** and **readv** return the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line [see **ioctl**(2) and **termio**(7I)] or if the number of bytes left in the file is less than *nbyte* or if the file is a pipe or a special file. A value of **0** is returned when an end-of-file has been reached.

read reads data previously written to a file. If any portion of an ordinary file prior to the end of file has not been written, **read** returns the number of bytes read as **0**. For example, the **lseek** routine allows the file pointer to be set beyond the end of existing data in the file. If additional data is written at this point, subsequent reads in the gap between the previous end of data and newly written data return bytes with a value of **0** until data is written into the gap.

A **read** or **readv** from a STREAMS [see **intro**(2TSOL)] file can operate in three different modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default is byte-stream mode. This default can be changed using the **I_SRDOPT ioctl**(2) request [see **streamio**(7I)], and can be tested with the **I_GRDOPT ioctl**(2) request.

In byte-stream mode, **read** and **readv** retrieve data from the stream until they have retrieved *nbyte* bytes or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

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

When attempting to read from a regular file with mandatory file / record locking set [see **chmod**(2TSOL)], when a write lock owned by another process exists on the segment of the file to be read,

- If **O_NDELAY** or **O_NONBLOCK** is set, **read** returns –**1** and sets **errno** to **EAGAIN**.
- If **O_NDELAY** and **O_NONBLOCK** are clear, **read** sleeps until the blocking record lock is removed.

When attempting to read from an empty pipe or FIFO,

- If no process has the pipe open for writing, **read** returns **0** to indicate end-of-file.
- If some process has the pipe open for writing and **O_NDELAY** is set, **read** returns **0.**
- If some process has the pipe open for writing and **O_NONBLOCK** is set, **read** returns –**1** and sets **errno** to **EAGAIN**.
- If **O_NDELAY** and **O_NONBLOCK** are clear, **read** blocks until data is written to the pipe or the pipe is closed by all processes that had opened the pipe for writing.

When attempting to read a file associated with a terminal that has no data currently available,

- If **O_NDELAY** is set, **read** returns **0**.
- If **O_NONBLOCK** is set, **read** returns –**1** and sets **errno** to **EAGAIN**.
- If **O_NDELAY** and **O_NONBLOCK** are clear, **read** blocks until data become available.

When attempting to read a file associated with a stream that is not a pipe nor FIFO nor terminal, and that has no data currently available,

- If **O_NDELAY** or **O_NONBLOCK** is set, **read** returns **−1** and sets **errno** to **EAGAIN**.
- If **O_NDELAY** and **O_NONBLOCK** are clear, **read** blocks until data becomes available.

When reading from a STREAMS file, handling of zero-byte messages is determined by the current read-mode setting. In byte-stream mode, **read** accepts data until it has read *nbyte* bytes or until there is no more data to read or until a zero-byte message block is encountered. **read** then returns the number of bytes read, and places the zero-byte message back on the stream to be retrieved by the next **read** or **getmsg**( ). [See **getmsg**(2).] In the two other modes, a zero-byte message returns a value of **0** and the message is removed from the stream. When a zero-byte message is read as the first message on a stream, a value of **0** is returned regardless of the **read** mode.

A **read** or **readv** from a STREAMS file returns the data in the message at the front of the stream-head read queue, regardless of the priority band of the message.

Normally, a **read** from a STREAMS file can process only messages with data and without control information.The **read** fails if a message containing control information is encountered at the stream head. This default action can be changed by placing the stream in either control-data mode or control-discard mode with the **I_SRDOPT ioctl**(2). In control-data mode, control messages are converted to data messages by **read**. In control-discard mode, control messages are discarded by **read**, but any data associated with the control messages is returned to the user.

readl, **preadl**, and **readvl** perform the same actions as **read**, **pread**, and **readv**, respectively, and additionally return in *label_p* the CMW label of the data read. The label returned is determined according to these conditions:

- If the descriptor refers to a regular file or FIFO, the sensitivity label portion of *label_p* is set to the sensitivity label associated with the file-system object.
- If the descriptor refers to a regular file, then the information label portion is set to the information label of the file.
- If the descriptor refers to a FIFO, then the information label portion is set to the information label that is associated with the data.

Later description details how the information label is associated with the data in a FIFO.

In all other respects, the **readl**, **preadl**, and **readvl** interfaces are analogous to the **read**, **pread**, and **readv** interfaces.

For the **read**, **readl**, **pread**, **preadl**, **readv**, and **readvl** interfaces, if the set of effective privileges of the calling process includes the **PRIV_PROC_NOFLOAT** privilege, then the information label of the calling process is not floated. Otherwise, the information label of the calling process is floated in this way:

- If the descriptor refers to a regular file, then the information label associated with the file is conjoined with the information label of the calling process.
- If the descriptor refers to a FIFO, then the information label is more complex. In a FIFO, like all conduits, each quantum of data has an associated set of attributes, including an information label. There may be multiple quanta of data in the FIFO at any one time.

If a read operation on a FIFO spans multiple quanta, then the information label associated with the data is the conjunction of the information labels associated with each quanta of data that was spanned. The information label associated with the data is conjoined with the information label of the calling process.

Under base Solaris, **read** normally allows a process to read the contents of directories on some local file systems. This functionality is not supported under Trusted Solaris. If the file descriptor refers to a directory, **read** will return **EISDIR**.

The last access time is updated only when the calling process has both mandatory read and write access to the file-system object. There is no privilege to override this restriction.

RETURN VALUES

On success, these functions return a nonnegative integer indicating the number of bytes actually read. Upon failure, the functions return −**1** and set **errno** is set to indicate the error.

ERRORS

read, **readl**, **pread**, **preadl**, **readv**, and **readvl** fail if any of these conditions is true:

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.
	No message is waiting to be read on a stream, and **O_NDELAY** or **O_NONBLOCK** was set.
EBADF	*fildes* is not a valid file descriptor open for reading.
EBADMSG	The message waiting to be read on a stream is not a data message.
EDEADLK	The read was about to go to sleep and cause a deadlock to occur.
EFAULT	*buf* points to an illegal address.
EINTR	A signal was caught during the read operation and no data was transferred.
EINVAL	Attempted to read from a stream linked to a multiplexor
EIO	A physical I / O error has occurred; or the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group of the process is orphaned.
EISDIR	*fildes* refers to a directory.
ENOLCK	The system record lock table was full, so the **read** or **readv** could not go to sleep until the blocking record lock was removed.
ENOLINK	*fildes* is on a remote machine but the link to that machine is no longer active.
ENXIO	The device associated with *fildes* is a block special or character special

file and the value of the file pointer is out of range.

In addition, **readv** may return one of these errors:

EFAULT	*iov* points to an illegal address.
EINVAL	*iovcnt* was less than or equal to **0,** or greater than or equal to **{IOV_MAX}**. [See **intro**(2TSOL) for a definition of **{IOV_MAX}**.] The sum of the **iov_len** values in the *iov* array overflowed an int.

In addition, **pread** fails and the file pointer remains unchanged if this true:

ESPIPE	*fildes* is associated with a pipe or FIFO.

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

In addition, **readl**, **preadl**, and **readvl** may set **errno** to

EFAULT	*label_p* points to an illegal address.

SUMMARY OF TRUSTED SOLARIS CHANGES

readl, **preadl**, and **readvl** return in the buffer referenced by *label_p* the CMW label associated with the data that was read.

The last access time is updated only when the calling process has both mandatory read and write access to the file-system object. There is no privilege to override this restriction.

For conduits, a sensitivity label is associated with each byte of data.

The calling process may assert the **PRIV_PROC_NOFLOAT** privilege to prevent the floating of its information label.

SEE ALSO

intro(2TSOL), **chmod**(2TSOL), **creat**(2TSOL), **dup**(2), **fcntl**(2TSOL), **getmsg**(2), **ioctl**(2), **open**(2TSOL), **pipe**(2), **streamio**(7I), **termio**(7I)

NAME

readlink – Read the value of a symbolic link

SYNOPSIS

#include <unistd.h>

int readlink(const char **path*, **void** **buf*, **size_t** *bufsiz***);**

DESCRIPTION

readlink( ) places the contents of the symbolic link to which *path* points in the buffer *buf*, which has size *bufsiz*. The contents of the link are not null-terminated when returned.

The information label of the link is unchanged. The information label of the calling process floats according to the information label of the symbolic link unless the calling process asserts the **PRIV_PROC_NOFLOAT** privilege.

RETURN VALUES

Upon successful completion, **readlink** returns the number of characters placed in the buffer. Upon failure, **readlink** returns –**1** and places an error code in **errno**.

ERRORS

readlink( ) fails and the buffer remains unchanged if any of these conditions is true:

EACCES	Search permission is denied for a component of the path prefix of *path*. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_SEARCH** and **PRIV_FILE_MAC_SEARCH**. Read permission is denied to the link. To override this restriction, the calling process may assert the **PRIV_FILE_MAC_READ** privilege.
EFAULT	*path* or *buf* points to an illegal address.
EINVAL	The named file is not a symbolic link.
EIO	An I/O error occurs while reading from or writing to the file system.
ELOOP	Too many symbolic links are encountered in translating *path*.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	The named file does not exist.
ENOSYS	The file system does not support symbolic links.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

The information label of the calling process is floated by the information label of the symbolic link unless the calling process asserts the **PRIV_PROC_NOFLOAT** privilege.

SEE ALSO

stat(2TSOL), **symlink**(2TSOL)

NAME

rename – Change the name of a file

SYNOPSIS

#include <stdio.h>

int rename(const char **_old_, const char** **_new_);**

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

The **rename** function changes the name of a file. *old* points to the path name of the file to be renamed. *new* points to the new path name 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 path name of a file that is not a directory, *new* must not point to the path name of a directory. If it exists, the link named by *new* 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 as *new* or the file referred to as *old* before the operation began.

If *old* points to the path name of a directory, *new* must not point to the path name of a file that is not a directory. If it exists, the directory named as *new* 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 named as *new* or the file named as *old* before the operation began. Thus, if *new* names an existing directory, it must be an empty directory.

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

If the directory containing *old* has the sticky bit set, at least one of these conditions 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 these conditions must be true:

- The user must own *new*.
- The user must own the directory containing *new*.
- *new* must be writable by the user.
- The user must be a privileged user.

If the link named by *new* exists, the link count of the file 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.

A single-level directory cannot be renamed (single-level directories are always contained in multilevel directories). A multilevel directory cannot be the new containing directory. There is no privilege to bypass these restrictions.

If the old containing directory is now empty, its information label is set to ADMIN_LOW.

RETURN VALUES

Upon successful completion, **rename** returns **0**. Upon failure, **rename** returns –**1** and sets **errno** to indicate an error.

ERRORS

rename fails and sets **errno** accordingly if any of these conditions is true:

EACCES	A component of either path prefix denies search permission; either the directory containing *old* or the directory containing *new* denies write permissions; or write permission is denied by a directory to which either *old* or *new* points. To bypass ownership restrictions, the calling process may assert one or more of these privileges: **PRIV_FILE_DAC_SEARCH**, **PRIV_FILE_MAC_SEARCH**, **PRIV_FILE_MAC_WRITE**, **PRIV_FILE_DAC_WRITE**, and **PRIV_FILE_OWNER**.
EBUSY	*new* is a directory and the mount point for a mounted file system.
EDQUOT	The directory in which 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 its parent directory (**..**).
EINVAL	*new* directory path name contains a path prefix that includes the *old* directory.
EISDIR	*new* points to a directory but *old* points to a file that is not a directory.
ELOOP	Too many symbolic links were encountered in translating the path name.
ENAMETOOLONG	The length of *old* or *new* exceeds **{PATH_MAX},** or a path name 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.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

A single-level directory cannot be renamed. A multilevel directory cannot be the new containing directory. There is no privilege to bypass these restrictions.

SEE ALSO

chmod(2TSOL), **link**(2TSOL), **unlink**(2TSOL)

NOTES

The system can deadlock if there is a loop in the file system graph. Such a loop occurs when an entry in directory **a** (say **a/name1**) is a hard link to directory **b**, and an entry in directory **b** (say **b/name2**) 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 to rename **b/name2** to **a/name1**, the system may deadlock attempting to lock both directories for modification. Use symbolic links instead of hard links for directories.

NAME

rmdir – Remove a directory

SYNOPSIS

#include <unistd.h>

int rmdir(const char **path***);**

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

rmdir( ) removes the directory to which **path** points. 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. A multilevel directory can be removed only when all its contained single-level directories are empty.

The information label of the calling process is unchanged.

RETURN VALUES

Upon successful completion, this function returns **0**. Upon failure, this function returns –**1** and sets **errno** to indicate the error.

ERRORS

The named directory is removed unless any of these conditions is true:

EACCES	Search permission is denied for a component of the path prefix. To override this restriction, the calling process must assert one or both of these privileges: **PRIV_FILE_DAC_SEARCH** and **PRIV_FILE_MAC_SEARCH**.
EACCES	Write permission is denied on the directory containing the directory to be removed. To bypass discretionary or mandatory write restrictions, the calling process must assert one or both of these privileges: **PRIV_FILE_DAC_WRITE** and **PRIV_FILE_MAC_WRITE**.
EACCES	If the containing directory has the the **S_ISVTX** variable set, the calling process must either be the owner of the containing directory or the directory to be deleted, or must have write access to the directory to be deleted. To override this restriction, the calling process may assert one or more of these privileges: **PRIV_FILE_MAC_WRITE**, **PRIV_FILE_DAC_WRITE**, and **PRIV_FILE_OWNER**.
EBUSY	The directory to be removed is the mount point for a mounted file system.
EEXIST	The directory contains entries other than “**.**” and “**..**”.

EFAULT	*path* points to an illegal address.
EINVAL	The directory to be removed is the current directory.
EINVAL	The final component of *path* is ‘‘.’’.
EIO	An I/O error occurred while accessing the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines and the file system does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	The named directory does not exist or is a null pathname.
ENOLINK	*path* points to a remote machine, but 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.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

A multilevel directory can be removed only when all its contained single-level directories are empty.

The information label of the calling process is unchanged.

SEE ALSO

mkdir(1), **rm**(1), **mkdir**(2TSOL)

NAME

secconf – get security configuration information

SYNOPSIS

#include <tsol/secconf.h>

long secconf(int *name***);**

AVAILABILITY

Available only on Trusted Solaris 2.x systems

DESCRIPTION

The **secconf( )** system call provides a method for an application to determine the current value of a configurable security system limit or option.

The *name* argument represents the system variable to be queried.

int *name*	**Variable Name**	**Description**
_TSOL_CLEAN_WINDOWS	tsol_clean_windows	Force cleaning of unused register windows before return from system call (SPARC architecture only)
_TSOL_ENABLE_IL	tsol_enable_il	ILs are allowed to be set (or displayed)
_TSOL_ENABLE_IL_FLOATING	tsol_enable_il_floating	Enables IL floating
_TSOL_FLOAT_SYSV_MSG_IL	tsol_float_sysv_msg_il	Enables IL floating on SVIPC message queues
_TSOL_FLOAT_SYSV_SEM_IL	tsol_float_sysv_sem_il	Enables IL floating on SVIPC semaphores
_TSOL_FLOAT_SYSV_SHM_IL	tsol_float_sysv_shm_il	Enables IL floating on SVIPC shared memory segments
_TSOL_FLUSH_BUFFERS	tsol_flush_buffers	Force flushing of file data blocks before inode updates
_TSOL_HIDE_UPGRADED_NAMES	tsol_hide_upgraded_names	Hide upgrade directory entries
_TSOL_PRIVS_DEBUG	tsol_privs_debug	Enables privilege debugging mode
_TSOL_RESET_IL_ON_EXEC	tsol_reset_il_on_exec	Reset IL to admin_low before exec

RETURN VALUES

If *name* is an invalid value, **secconf**( ) will return **–1** and set **errno** to indicate the error. If **secconf**( ) fails due to a value of **name** that is not defined on the system, the call will return a value of **–1** without changing the value of errno.

ERRORS

The function will return the following errors:

EINVAL	The parameter *name* is unknown.

FILES

system(4)	system configuration information file

SEE ALSO

pathconf(2), **sysconf**(3C)

NAME

semctl – Semaphore-control operations

SYNOPSIS

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

int semctl(int *semid***, int** *semnum***, int** *cmd***, ...);**

DESCRIPTION

semctl( ) provides a variety of semaphore-control operations as specified by *cmd*. The fourth argument is optional, depending upon the operation requested. If required, the fourth argument is of type **union semun**, which must be explicitly declared by the application program.

```
union semun {
        int val;
        struct semid_ds *buf;
        ushort *array
} arg;
```

The permission required for a semaphore operation is given as {*token*}, where *token* is the type of permission needed. The types of permission are interpreted as follows:

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

The commands described hereafter as [READ] operations all require that the calling process have discretionary read access to the data structure referenced by *semid*, or that the effective privilege set of the process include **PRIV_IPC_DAC_READ**. Likewise, the commands described as [ALTER] operations all require that the calling process have discretionary write access to the data structure referenced by *semid*, or that the effective privilege set of the process include **PRIV_IPC_DAC_WRITE**.

If the sensitivity label of the object does not match the sensitivity label of the calling process, then the process must have these privileges asserted: **PRIV_IPC_MAC_READ** for [READ] operations; **PRIV_IPC_MAC_WRITE** for [ALTER] operations.

See the **Semaphore Operation Permissions** subsection of the **DEFINITIONS** section of **intro**(2TSOL) for more information. These 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**(2TSOL).] {READ}

SETVAL Set the value of **semval** to *arg*.**val**. {ALTER}. When this command is successfully executed, the **semadj** value corresponding to the specified semaphore in all processes is cleared.

GETPID Return the value of **(int) sempid**. {READ}

GETNCNT Return the value of **semncnt**. {READ}

GETZCNT Return the value of **semzcnt**. {READ}

The following operations return and set, respectively, every **semval** in the set of semaphores:

GETALL Place **semval**s into array to which *arg*.**array** points. {READ}

SETALL Set **semval**s according to the array to which *arg*.**array** points. {ALTER}. When this *cmd* is successfully executed, the **semadj** values corresponding to each specified semaphore in all processes are cleared.

If information-label floating is enabled on this system, the **SETALL** and **SETVAL** operations may float the information label of the semaphore group unless the **PRIV_IPC_NOFLOAT** privilege is asserted. Likewise, the **GETALL** and **GETVAL** operations may float the information label of the process unless the **PRIV_PROC_NOFLOAT** privilege is asserted.

These operations are also available:

IPC_STAT Place the current value of each member of the data structure associated with *semid* into the structure to which *arg*.**buf** points. The contents of this structure are defined in **intro**(2TSOL). {READ}

IPC_SET Set the value of these members of the data structure associated with *semid* to the corresponding value found in the structure to which *arg*.**buf** points:

```
sem_perm.uid
sem_perm.gid
sem_perm.mode     /* only access permission bits */
```

This command can be executed only by a process that either has an effective user ID equal to **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with **semid**, or has the **PRIV_IPC_OWNER** privilege in its set of effective privileges. In addition, the process must either have mandatory write access to the Semaphore set or have asserted the **PRIV_IPC_MAC_WRITE** privilege.

IPC_RMID Remove from the system the semaphore identifier specified by *semid* and destroy the set of semaphores and data structure associated with that identifier. This command can be executed only by a process that either has an effective user ID equal to **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with semid, or has the **PRIV_IPC_OWNER** privilege asserted. In addition, the process must also have mandatory write access to the Semaphore set or have asserted the **PRIV_IPC_MAC_WRITE** privilege.

RETURN VALUES

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

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**; failures return –**1** and set **errno** to indicate the error.

ERRORS

semctl fails if any of these conditions is true:

EACCES	Operation permission is denied to the calling process [see **Intro**(2TSOL)], and the process lacks the appropriate privilege override(s) in its set of effective privileges.
EINVAL	*semid* is not a valid semaphore identifier.
	semnum is less than **0** or greater than **sem_nsems** –**1**.
	cmd is not a valid command.
	cmd is **IPC_SET** and **sem_perm.uid** or **sem_perm.gid** is not valid.
	For **GETVAL** or **GETALL**, the operation would float the information label of the process above its sensitivity label.
	For **SETVAL** or **SETALL**, the operation would float the information label of the object above its sensitivity label.
EPERM	*cmd* is equal to **IPC_RMID** or **IPC_SET** and the effective user of the calling process is not equal to the value of **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with *semid*, and the appropriate privilege is not asserted.
EOVERFLOW	*cmd* is **IPC_STAT** and *uid* or *gid* is too large to be stored in the structure to which *arg.buf* points.
ERANGE	*cmd* is **SETVAL** or **SETALL** and the value to which **semval** is to be set is greater than the system-imposed maximum.

SUMMARY OF TRUSTED SOLARIS DIFFERENCES

Appropriate privilege is required to override access checks.

SEE ALSO

ipcs(1TSOL), **intro**(2TSOL), **semget**(2TSOL), **semop**(2TSOL)

NAME

semget, semgetl – 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***);**

cc [*flag* **...]** *file* **... -ltsol [** *library* **...]**
#include <sys/tsol/ipcl.h>

int semgetl(key_t *key***, int** *nsems***, int** *semflg***, const bslabel_t** **slabel***);**

DESCRIPTION

A semaphore structure is identified by a unique combination of key and sensitivity label. This qualification of keys by sensitivity labels allows applications that use semaphore structures to be run at multiple process sensitivity labels without inadvertently sharing data.

semget( ) returns the semaphore identifier associated with the union of key and the sensitivity label of the calling process. **semgetl**( ) returns the semaphore-structure identifier associated with the union of *key* and *slabel*. If the value of *slabel* does not match the sensitivity label of the calling process, then the effective privilege set of the process must include both **PRIV_IPC_MAC_READ** and **PRIV_IPC_MAC_WRITE**.

If discretionary read / write access as specified by the low-order 9 bits of *semflg* is denied to the calling process, **semget** and **semgetl** require one or both of these privileges: **PRIV_IPC_DAC_READ** and **PRIV_IPC_DAC_WRITE**.

A semaphore identifier and associated data structure and set containing *nsems* semaphores [see **intro**(2)] are created for *key* if one of these conditions 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** and **sem_perm.uid** are set equal to the effective user ID, and **sem_perm.cgid** and **sem_perm.gid** are set equal to the effective group ID 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.

If information labels are enabled on this system, the information label on a new semaphore set is ADMIN_LOW.

RETURN VALUES

Upon successful completion, **semget** and **semgetl** return a nonnegative integer, namely a semaphore identifier. Upon failure, **semget** and **semgetl** return –**1** and sets **errno** to indicate the error.

ERRORS

semget and **semgetl** fail if any of these conditions is true:

EACCES A semaphore-structure identifier exists for the union of key and sensitivity label, but operation permission [see intro(2)] as specified by the low-order 9 bits of *semflg* would not be granted; or the sensitivity label check did not pass, and the calling process does not have the appropriate privilege override(s) in its set of effective privileges.

EEXIST A semaphore identifier exists for *key* but (*semflg*&**IPC_CREAT**) and (*semflg*&**IPC_EXCL**) are both true.

EFAULT *slabel* points to an illegal address.

EINVAL The label to which *slabel* points is not a valid sensitivity label.

nsems is either less than or equal to zero or greater than the system-imposed limit.

A semaphore identifier exists for *key*, but the number of semaphores in the set associated with it is less than *nsems*, and *nsems* is not equal to zero.

ENOENT A semaphore-structure identifier does not exist for the union of *key* and sensitivity label, and (*semflg*&**IPC_CREAT**) is false.

ENOSPC A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphore identifiers systemwide would be exceeded.

A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores systemwide would be exceeded.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

Sensitivity labels are used together with key to determine semaphore-group identifiers.

SEE ALSO

ipcs(1TSOL), **ipcrm**(1TSOL), **intro**(2TSOL), **semctl**(2TSOL), **semop**(2TSOL), **stdipc**(3C)

NAME

semop, semopl – 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***);**

cc [*flag* **...]** *file* **... -ltsol [** *library* **...]**
#include <sys/tsol/ipcl.h>

int semopl(int *semid***, struct sembuf** ∗*sops***, size_t** *nsops***,**
bilabel_t ∗*ilabel***);**

DESCRIPTION

semop( ) is used to perform atomically an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by *semid*. *sops* is a pointer to the array of semaphore-operation structures. *nsops* is the number of such structures in the array. The contents of each structure includes these members:

```
short    sem_num;    /* semaphore number */
short    sem_op;     /* semaphore operation */
short    sem_flg;    /* operation flags */
```

semop( ) uses the current information label of the process to label a semaphore group. **semopl**( ) is identical to **semop**( ) but allows the calling process to label a semaphore group with a particular information label as specified by *ilabel*. If *ilabel* does not match the current information label of the process, **semopl** requires that the effective privilege set of the calling process include these privileges:

- **PRIV_IPC_UPGRADE_IL** if *ilabel* specifies an upgrade
- **PRIV_IPC_DOWNGRADE_IL** if *ilabel* specifies a downgrade

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**(2TSOL) for more information.

sem_op specifies the {ALTER} token if its value is negative or positive, and the {READ} token if its value is zero. What may occur depends on the value of *sem_op*:

sem_op is a negative integer; {ALTER}

- If **semval** [see **intro**(2TSOL)] is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from **semval**. Also, if (*sem_flg***&SEM_UNDO**) is true, the absolute value of *sem_op* is added to the **semadj** value of the calling process [see **exit**(2)] for the specified semaphore.
- If **semval** is less than the absolute value of *sem_op* and (*sem_flg***&IPC_NOWAIT**) is true, **semop**( ) returns immediately.
- If **semval** is less than the absolute value of *sem_op* and (*sem_flg***&IPC_NOWAIT**) is false, **semop**( ) increments the **semncnt** associated with the specified semaphore and suspends execution of the calling process until one of these conditions occurs:
  - **semval** becomes greater than or equal to the absolute value of *sem_op*. When this condition 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**(2TSOL).] When this removal occurs, **errno** is set equal to **EIDRM**, and a value of −**1** is returned.
  - The calling process receives a signal that is to be caught. When this reception occurs, the value of **semncnt** associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in **signal**(3C).

sem_op is a positive integer; {ALTER}

- The value of *sem_op* is added to **semval**; and if (*sem_flg***&SEM_UNDO**) is true, the value of *sem_op* is subtracted from the calling process's **semadj** value for the specified semaphore.

sem_op is zero; {READ}

- If **semval** is zero, **semop**( ) returns immediately.
- If **semval** is not equal to zero and (*sem_flg***&IPC_NOWAIT**) is true, **semop**( ) returns immediately.
- If **semval** is not equal to zero and (*sem_flg***&IPC_NOWAIT**) is false, **semop**( ) increments the **semzcnt** associated with the specified semaphore and suspends execution of the calling process until one of the conditions occurs:
  - **semval** becomes zero, at which time the value of **semzcnt** associated with the specified semaphore is decremented.
  - The *semid* for which the calling process is awaiting action is removed from the system. When this removal occurs, **errno** is set equal to **EIDRM**, and a value of −**1** is returned.
  - The calling process receives a signal that is to be caught. When this reception

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).

If *sem_op* is zero {READ}, the process must have discretionary and mandatory read access to the semaphore structure to which *semid* refers. Overriding these checks requires that the effective privilege set of the process include one or both of these privileges as necessary: **PRIV_IPC_DAC_READ** and **PRIV_IPC_MAC_READ**.

If *sem_op* is a positive or a negative number {ALTER}, the process must have discretionary and mandatory write access to the semaphore structure to which *semid* refers. Overriding these checks requires that the effective privilege set of the process include one or both of these privileges as necessary: **PRIV_IPC_DAC_WRITE** and **PRIV_IPC_MAC_WRITE**.

Unlike messages within a message queue, the individual semaphores within a semaphore structure are not labeled with an information label.

If information label floating is enabled on this system, this system call may cause the information label of the subject or the object to float. For **semop**( )-{ALTER}, the information label of the semaphore group is combined with the information label of the subject to "float up" the semaphore group's information label unless the subject has asserted the **PRIV_IPC_NOFLOAT** privilege. For **semopl**, the information label of the semaphore group is combined with the information label supplied by the caller. The supplied information label must be dominated by the sensitivity label of the semaphore set. If the supplied information label dominates but is not equal to the information label of the subject (an upgrade), the subject must assert the **PRIV_IPC_UPGRADE_IL** privilege. If the supplied information label does not dominate the information label of the subject (a downgrade), the subject must assert the **PRIV_IPC_DOWNGRADE_IL** privilege. If *sem_op* is zero, the information label of the semaphore is unchanged.

If *sem_op* is zero-{READ}, then the information label of the process is combined with the information label of the semaphore group to "float up" the information label of the subject unless the subject has asserted **PRIV_PROC_NOFLOAT**.

RETURN VALUES

Upon successful completion, **semop** and **semopl** return a value of zero. Upon failure, **semop** and **semopl** return a value of **–1** and sets **errno** to indicate the error.

ERRORS

semop and **semopl** fail if any of these conditions is true for any of the semaphore operations specified by *sops*:

E2BIG *nsops* is greater than the system-imposed maximum.

EACCES Operation permission is denied to the calling process ([see intro(2)], and the calling process does not have the appropriate privilege(s) in its set of effective privileges.

EAGAIN The operation would result in suspension of the calling process but (*sem_flg*&**IPC_NOWAIT**) is true.

EFAULT *sops* points to an illegal address.

The address to which *ilabel* points is illegal.

EFBIG	*sem_num* is less than zero or greater than or equal to the number of semaphores in the set associated with *semid*.
EIDRM	**semop( )** A *semid* was removed from the system.
EINTR	A signal was received.
EINVAL	*semid* is not a valid semaphore identifier, or the number of individual semaphores for which the calling process requests a **SEM_UNDO** would exceed the limit.
	This operation would cause the information label of the semaphore group to dominate the sensitivity label of the group, and the calling process does not have the appropriate privilege override(s) in its set of effective privileges.
	This operation would cause the information label of the calling process to float above its sensitivity label, and the calling process does not have the appropriate privilege override in its set of effective privileges.
	The label to which *ilabel* points is not a valid information label.
ENOSPC	The limit on the number of individual processes requesting a **SEM_UNDO** would be exceeded.
EPERM	This call is trying either to upgrade or to downgrade the new information label from the IL of the process but is not suitably privileged.
ERANGE	An operation would cause a **semval** or a **semadj** value to overflow the system-imposed limit.

Upon successful completion, the value of **sempid** for each semaphore specified in the array to which *sops* points is set equal to the process ID of the calling process.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

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

NAME

setclearance – Set process clearance

SYNOPSIS

#include <tsol/label.h>

int setclearance(bclear_t ∗*clearance_p***);**

DESCRIPTION

setclearance( ) is used to set the clearance for the calling process provided it has the **PRIV_PROC_SETCLR** privilege in its set of effective privileges. **setclearance**( ) verifies that the information pointed to by *clearance_p* is formatted correctly, and that the resulting clearance will dominate the sensitivity label of the process.

RETURN VALUES

setclearance( ) returns **0** if successful. If not, **setclearance** returns –**1** and sets **errno** to indicate the error.

ERRORS

setclearance( ) fails and does not set the process clearance if any of these conditions prevails:

EFAULT	The *clearance_p* argument points to an invalid address.
EINVAL	The *clearance_p* argument does not point to a properly formatted clearance.
EINVAL	The clearance pointed to by *clearance_p* does not dominate the process sensitivity label.
EPERM	The calling process does not have the necessary privilege (**PRIV_PROC_SETCLR**) to set the clearance.

SEE ALSO

getclearance(2TSOL)

NAME

setcmwlabel, fsetcmwlabel, lsetcmwlabel – Set CMW label of a file

SYNOPSIS

#include <tsol/label.h>

int setcmwlabel(const char * *path*, **const bclabel_t** * *label_p*
 const setting_flag_t *flag***);**

int fsetcmwlabel(int *fd*, **const bclabel_t** * *label_p*,
 const setting_flag_t *flag***);**

int lsetcmwlabel(const char * *path*, **const bclabel_t** * *label_p*,
 const setting_flag_t *flag***);**

DESCRIPTION

The file that is named by *path* or referred to by *fd* has its CMW label changed as specified provided the file resides on a UFS or TMPFS file system.

If *flag* equals **SETCL_ALL**, then both parts of the file's CMW label are to be set and the following checks must be made:

- The sensitivity label must dominate the information label.
- The sensitivity label of *label_p* must be in the sensitivity label range of the containing file system.
- If the information label of *label_p* does not dominate the existing information label (a downgrade) and the calling process is not the owner of the file, then the calling process must have **PRIV_FILE_OWNER** in its set of effective privileges.
- If the sensitivity label of *label_p* equals the existing sensitivity label, then neither **PRIV_FILE_UPGRADE_SL** nor **PRIV_FILE_DOWNGRADE_SL** is required.
- If the sensitivity label of *label_p* dominates but does not equal the existing sensitivity label (an upgrade), then the calling process must have **PRIV_FILE_UPGRADE_SL** in its set of effective privileges.
- If the sensitivity label of *label_p* does not dominate the existing sensitivity label (a downgrade), then the calling process must have **PRIV_FILE_DOWNGRADE_SL** in its set of effective privileges.
- If the sensitivity label operation is a downgrade and the calling process is not the owner of the file, then the calling process must have **PRIV_FILE_OWNER** in its set of effective privileges.

If *flag* is equal to **SETCL_IL**, then the information label of the file's CMW label is to be set and the following checks must be made:

- The existing sensitivity label must dominate the information label of *label_p*.
- If the information label of *label_p* equals the existing information label, then neither **PRIV_FILE_UPGRADE_IL** nor **PRIV_FILE_DOWNGRADE_IL** is required.
- If the information label of *label_p* dominates but does not equal the existing information label (an upgrade), then the calling process must have **PRIV_FILE_UPGRADE_IL** in its set of effective privileges.
- If the information label of *label_p* does not dominate the existing information label (a

downgrade), then the calling process must have **PRIV_FILE_DOWNGRADE_IL** in its set of effective privileges.

- If the operation is an information label downgrade and the calling process is not the owner of the file, then the calling process must have **PRIV_FILE_OWNER** in its set of effective privileges.

If *flag* equals **SETCL_SL**, then the sensitivity label of the file's CMW label is to be set and the following checks must be made:

- The sensitivity label of *label_p* must dominate the existing information label.
- The sensitivity label of *label_p* must be in the sensitivity label range of the containing file system.
- If the sensitivity label of *label_p* equals the existing sensitivity label, then neither **PRIV_FILE_UPGRADE_SL** nor **PRIV_FILE_DOWNGRADE_SL** is required.
- If the sensitivity label of *label_p* dominates but does not equal the existing sensitivity label (an upgrade), then the calling process must have **PRIV_FILE_UPGRADE_SL** in its set of effective privileges.
- If the sensitivity label of *label_p* does not dominate the existing sensitivity label (a downgrade), then the calling process must have **PRIV_FILE_DOWNGRADE_SL** in its set of effective privileges.
- If the operation is a sensitivity label downgrade and the calling process is not the owner of the file, then the calling process must have **PRIV_FILE_OWNER** in its set of effective privileges.

There are several checks that are applicable if either the information label or the sensitivity label is being changed:

- The calling process must have discretionary write access to the file.
- If there is an open file descriptor reference to the file, then the calling process must have **PRIV_PROC_TRANQUIL** in its set of effective privileges.

setcmwlabel( ) and **lsetcmwlabel( )** function identically except when the final component is a symbolic link. If the final component is a symbolic link, **lsetcmwlabel( )** sets the CMW label of the symbolic link, but **setcmwlabel( )** sets the CMW label of the object referred to by the symbolic link.

NOTES

If the sensitivity label is being set, then the calling process is responsible for verifying that sensitivity label is within the accreditation range of the system.

RETURN VALUES

setcmwlabel( ) , **fsetcmwlabel( )** , and **lsetcmwlabel( )** return **0** if successful. If not, they return **−1** and set **errno** to indicate the error.

ERRORS

setcmwlabel( ) , and **lsetcmwlabel( )** fail and the file is unchanged if any of these conditions prevails:

EACCES	Search permission is denied for a component of the path prefix of *path*.
	The calling process does not have mandatory write access to the final

component of *path* because the sensitivity label of the final component of *path* does not dominate the sensitivity label of the calling process and the calling process does not have **PRIV_FILE_MAC_WRITE** in its set of effective privileges.

The calling process does not have discretionary write access to the final component of *path*.

EBUSY There is an open file descriptor reference to the final component of *path* and the calling process does not have **PRIV_PROC_TRANQUIL** in its set of effective privileges.

EFAULT *path* or *label_p* points outside the allocated address space of the process.

EINVAL *path* does not reside on a UFS or TMPFS file system.

flag is **SETCL_ALL** and the sensitivity label of *label_p* does not dominate the information label of *label_p*.

flag is **SETCL_IL** and the existing sensitivity label does not dominate the information label of *label_p*.

flag is **SETCL_SL** and the sensitivity label of *label_p* does not dominate the existing information label.

The sensitivity label of *label_p* is not in the sensitivity label range of the containing file system.

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

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

ENAMETOOLONG

The length of the path argument exceeds {PATH_MAX}.

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

ENOENT The file referred to by *path* does not exist.

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

EPERM The calling process does not have mandatory write access to the final component of *path* because the sensitivity label of the final component of *path* is outside the clearance of the calling process and the calling process does not have **PRIV_FILE_MAC_WRITE** in its set of effective privileges.

The calling process attempted to upgrade the information label associated with the final component of *path* but did not have **PRIV_FILE_UPGRADE_IL** in its set of effective privileges.

The calling process attempted to downgrade the information label associated with the final component of *path* but did not have **PRIV_FILE_DOWNGRADE_IL** in its set of effective privileges.

A calling process that is not the owner of the file attempted to downgrade the information label or sensitivity label associated with the final

	component of *path* but did not have **PRIV_FILE_OWNER** in its set of effective privileges.
	The calling process attempted to upgrade the sensitivity label associated with the final component of *path* but did not have **PRIV_FILE_UPGRADE_SL** in its set of effective privileges.
	The calling process attempted to downgrade the sensitivity label associated with the final component of *path* but did not have **PRIV_FILE_DOWNGRADE_SL** in its set of effective privileges.
EROFS	The file referred to by *path* resides on a read-only file system.

fsetcmwlabel( ) fails if if any of these conditions prevails:

EACCES	The descriptor is open only for reading, the calling process does not have mandatory write access to the object referred to by the descriptor because the sensitivity label of the object does not dominate the sensitivity label of the calling process, and the calling process does not have **PRIV_FILE_MAC_WRITE** in its set of effective privileges.
	The calling process does not have discretionary write access to the object referred to by the descriptor.
EBADF	*fd* does not refer to a valid descriptor.
EBUSY	There is an open file descriptor reference to the object referred to by the descriptor and the calling process does not have **PRIV_PROC_TRANQUIL** in its set of effective privileges.
EFAULT	*label_p* points outside the allocated address space of the process.
EINVAL	*fd* refers to a socket, not a file.
	fd does not refer to a file on a UFS or TMPFS file system.
	flag is **SETCL_ALL** and the sensitivity label of *label_p* does not dominate the information label of *label_p*.
	flag is **SETCL_IL** and the existing sensitivity label does not dominate the information label of *label_p*.
	flag is **SETCL_SL** and the sensitivity label of *label_p* does not dominate the existing information label.
	The sensitivity label of *label_p* is not in the sensitivity label range of the containing file system.
EIO	An I/O error occurred while reading from or writing to the file system.
EPERM	The calling process attempted to upgrade the information label associated with the file but did not have **PRIV_FILE_UPGRADE_IL** in its set of effective privileges.
	The calling process attempted to downgrade the information label associated with the file but did not have **PRIV_FILE_DOWNGRADE_IL** in its set of effective privileges.

The calling process is not the owner of the file, attempted to downgrade the information label or sensitivity label associated with the file, but did not have **PRIV_FILE_OWNER** in its set of effective privileges.

The calling process attempted to upgrade the sensitivity label associated with the file but did not have **PRIV_FILE_UPGRADE_SL** in its set of effective privileges.

The calling process attempted to downgrade the sensitivity label associated with the file but did not have **PRIV_FILE_DOWNGRADE_SL** in its set of effective privileges.

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

SEE ALSO

getcmwfsrange(2TSOL), **getcmwlabel**(2TSOL)

NAME

setcmwplabel – Set process CMW label

SYNOPSIS

cc [*flag* ...] *file* ... **–ltsol** [*library* ...]

#include <tsol/label.h>

int setcmwplabel(bclabel_t **label_p*, **setting_flag_t** *flag***)**

DESCRIPTION

setcmwplabel( ) sets the information label, the sensitivity label, or both labels for the process making the call. The *flag* argument identifies which label(s) to set:

SETCL_ALL	Set the entire CMW label of the process.
SETCL_SL	Set only the sensitivity label.
SETCL_IL	Set only the information label.

setcmwplabel verifies that the CMW label to which *label_p* points is formatted correctly and that the resulting label would satisfy the requirement that the clearance must dominate the sensitivity label, which must dominate the information label of the process.

When *flag* limits the setting to a single portion of the CMW label, **setcmwplabel** ignores the other value in *label_p*. If the specified values for sensitivity label or information label do not match current values of the process, the set of effective privileges of the calling process must include **PRIV_PROC_SETSL** or **PRIV_PROC_SETIL**, respectively.

RETURN VALUES

Upon success, **setcmwplabel** returns **0**. Upon failure, **setcmwplabel** returns –**1** and sets **errno** to indicate the error

ERRORS

setcmwplabel fails and does not set the process CMW label if any of these conditions is true:

EFAULT	The *label_p* argument points to an invalid address.
EINVAL	The *label_p* argument points to an improperly formatted label.
	The *label_p* argument and the *flag* argument would cause the process information label not to be dominated by the process sensitivity label.
	The *label_p* argument and the *flag* argument would cause the process sensitivity label not to be dominated by the clearance.
EPERM	The calling process lacks the **PRIV_PROC_SETIL** privilege necessary to set the information label specified by *flag*.
	The calling process lacks the **PRIV_PROC_SETSL** privilege necessary to set the sensitivity label specified by *flag*.

SEE ALSO

getcmwplabel(2TSOL)

NAME

setregid – set real and effective group IDs

SYNOPSIS

#include <unistd.h>

int setregid(gid_t *rgid***, gid_t** *egid***);**

DESCRIPTION

setregid( ) is used to set the real and effective group IDs of the calling process. If *rgid* is –**1**, the real GID is not changed; if *egid* is –**1**, the effective GID is not changed. The real and effective GIDs may be set to different values in the same call.

If the calling process has the **PRIV_PROC_SETID** privilege, the real GID and the effective GID can be set to any legal value.

If the calling process does not have the **PRIV_PROC_SETID** privilege, either the real GID can be set to the saved setGID from **execve**(2), or the effective GID can either be set to the saved setGID or the real GID. Note: if a setGID process sets its effective GID to its real GID, it can still set its effective GID back to the saved setGID.

In either case, if the real GID is being changed (that is, if *rgid* is not –**1**), or the effective GID is being changed to a value not equal to the real GID, the saved setGID is set equal to the new effective GID.

RETURN VALUES

setregid( ) returns:

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

ERRORS

setregid( ) will fail and neither of the group IDs will be changed if:

EINVAL	The value of *rgid* or *egid* is less than **0** or greater than **USHRT_MAX** (defined in **<limits.h>**).
EPERM	The calling process does not have the **PRIV_PROC_SETID** privilege and a change other than changing the real GID to the saved setGID, or changing the effective GID to the real GID or the saved GID, was specified.

SUMMARY OF TRUSTED SOLARIS CHANGES

The super-user check in base Solaris is replaced by privilege check in Trusted Solaris.

SEE ALSO

execve(2TSOL), **getgid**(2), **setreuid**(2TSOL), **setuid**(2TSOL),

NAME

setreuid – set real and effective user IDs

SYNOPSIS

#include <unistd.h>

int setreuid(uid_t *ruid***, uid_t** *euid***);**

DESCRIPTION

setreuid( ) is used to set the real and effective user IDs of the calling process. If *ruid* is –**1**, the real user ID is not changed; if *euid* is –**1**, the effective user ID is not changed. The real and effective user IDs may be set to different values in the same call.

If the calling process has the **PRIV_PROC_SETID** privilege, the real user ID and the effective user ID can be set to any legal value.

If the calling process does not have the **PRIV_PROC_SETID** privilege, either the real user ID can be set to the effective user ID, or the effective user ID can either be set to the saved set-user ID from **execve**(2) or the real user ID. Note: if a set-UID process sets its effective user ID to its real user ID, it can still set its effective user ID back to the saved set-user ID.

In either case, if the real user ID is being changed (that is, if *ruid* is not –**1**), or the effective user ID is being changed to a value not equal to the real user ID, the saved set-user ID is set equal to the new effective user ID.

RETURN VALUES

setreuid( ) returns:

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

ERRORS

setreuid( ) will fail and neither of the user IDs will be changed if:

EINVAL	The value of *ruid* or *euid* is less than **0** or greater than **USHRT_MAX** (defined in **<limits.h>**).
EPERM	The calling process does not have the **PRIV_PROC_SETID** privilege 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.

SUMMARY OF TRUSTED SOLARIS CHANGES

The super-user check in base Solaris is replaced by privilege check in Trusted Solaris.

SEE ALSO

execve(2TSOL), **getuid**(2), **setregid**(2TSOL), **setuid**(2TSOL)

NAME

setuid, setegid, seteuid, setgid – Set user and group IDs

SYNOPSIS

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

int setuid(uid_t *uid***);**

int setegid(gid_t *egid***);**

int seteuid(uid_t *euid***);**

int setgid(gid_t *gid***);**

MT-LEVEL

setuid( )and**setgid( )** are Async-Signal-Safe.

DESCRIPTION

The **setuid( )** function sets the real user ID, effective user ID, and saved user ID of the calling process. The **setgid** function sets the real group ID, effective group ID, and saved group ID of the calling process. The **setegid** and **seteuid** functions set the effective group and the effective user IDs respectively for the calling process. See **intro**(2TSOL) for more information on real, effective, and saved user and group IDs.

At login time, the real user ID, effective user ID, and saved user ID of the login process are set to the login ID of the user responsible for the creation of the process. The same is true for the real, effective, and saved group IDs; they are set to the group ID of the user responsible for the creation of the process.

When a process calls **exec**(2TSOL) 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 process calling **setuid** has the **PRIV_PROC_SETID** privilege, the real, effective, and saved user IDs are set to the *uid* parameter.

If the process calling **setuid** does not have the **PRIV_PROC_SETID** privilege, 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 new user ID differs from the initial user ID under which this program began execution, the saved privilege set is replaced by the effective privilege set; and the effective privilege set is cleared. If the new user ID matches the initial user ID under which this program began execution, the saved privilege set replaces the effective privilege set.

If the process calling **setgid** has the **PRIV_PROC_SETID** privilege, the real, effective, and saved group IDs are set to the *gid* parameter.

If the process calling **setgid** does not have the **PRIV_PROC_SETID** privilege, 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, these functions return a value of **0**. Otherwise, they return a value of –**1** and set **errno** to indicate the error.

ERRORS

setuid and **setgid** fail if either of these conditions is true:

EINVAL The *uid* or *gid* is out of range.

EPERM For **setuid** and **seteuid**, the calling process does not have **PRIV_PROC_SETID** in its effective set of privileges, and the *uid* parameter does not match either the real or saved user IDs.

For **setgid** and **setegid**, the calling process does not have **PRIV_PROC_SETID** in its effective set of privileges, and the *gid* parameter does not match either the real or the saved group ID.

SUMMARY OF TRUSTED SOLARIS CHANGES

A check for **PRIV_PROC_SETID** replaces super-user checks in the base Solaris.

SEE ALSO

intro(2TSOL), **exec**(2TSOL), **getgroups**(2TSOL), **getuid**(2), **stat**(5)

NAME

shmctl – Shared-memory control operations

SYNOPSIS

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

int shmctl(int *shmid***, int** *cmd***, struct shmid_ds** *∗buf***);**

DESCRIPTION

shmctl( ) provides a variety of shared-memory control operations as specified by *cmd*. The permission required for a shared-memory control operation is given as {*token*}, where *token* is the type of permission needed. The types of permission are interpreted as follows:

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

See the *Shared Memory Operation Permissions* section of **intro**(2TSOL) for more information.

The following operations require the specified tokens:

IPC_STAT Place the current value of each member of the data structure associated with *shmid* into the structure to which *buf* points. The contents of this structure are defined in **intro**(2TSOL). {READ}

The calling process must either have mandatory read access to the shared-memory segment or have asserted the **PRIV_IPC_MAC_READ** privilege, and either have discretionary read access to the data structure or have **PRIV_IPC_DAC_READ** in its set of effective privileges.

IPC_SET Set the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure to which *buf* points:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode          /* only access permission bits */
```

This command can be executed only by a process that either has an effective user ID equal to **sem_perm.cuid** or **sem_perm.uid** in the data structure associated with semid, or has **PRIV_IPC_OWNER** in its set of effective privileges. In addition, the process must either have mandatory write access to the semaphore set or have asserted the **PRIV_IPC_MAC_WRITE** privilege.

IPC_RMID Remove from the system the shared-memory identifier specified by *shmid* and destroy the shared-memory segment and data structure associated with the identifier. This command can be executed only by a process that either has an effective user ID equal to **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with *shmid*, or has **PRIV_IPC_OWNER** in its set of effective privileges. In addition, the process must either have mandatory write access to the shared memory segment or have asserted the **PRIV_IPC_MAC_WRITE** privilege.

SHM_LOCK Lock the shared-memory segment specified by *shmid* in memory. This command can be executed only by a process that has discretionary and mandatory read access (or the appropriate privilege override) and also has **PRIV_SYS_CONFIG** in its effective privilege set.

SHM_UNLOCK Unlock the shared-memory segment specified by *shmid.* This command can be executed only by a process that has discretionary and mandatory read access (or the appropriate privilege override) and also has **PRIV_SYS_CONFIG** in its effective privilege set.

RETURN VALUES

Upon successful completion, **shmctl** returns a value of **0**. Upon failure, **shmctl** returns a value of –**1** and sets **errno** to indicate the error.

ERRORS

shmctl( ) fails if any of these conditions is true:

EACCES	*cmd* is equal to **IPC_STAT**. {READ} operation permission is denied to the calling process, and the calling process does not have the appropriate privilege(s) in its set of effective privileges.
EFAULT	*buf* points to an illegal address.
EINVAL	*shmid* is not a valid shared-memory identifier.
	cmd is not a valid command.
	cmd is **IPC_SET** and **shm_perm.uid** or **shm_perm.gid** is not valid.
ENOMEM	*cmd* is equal to **SHM_LOCK** and there is not enough memory.
EOVERFLOW	*cmd* is **IPC_STAT** and *uid* or *gid* is too large to be stored in the structure to which *buf* points.
EPERM	*cmd* is equal to **IPC_RMID** or **IPC_SET**. The effective user ID of the calling process does not match the value of **shm_perm.cuid** or **shm_perm.uid** in the data structure associated with shmid; or the mandatory access check failed; and the calling process does not have the appropriate privilege overrides(s) in its set of effective privileges.
	cmd is equal to **SHM_LOCK** or **SHM_UNLOCK** and **PRIV_SYS_CONFIG** is not in the effective privilege set of the process.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO **ipcs**(1TSOL), **intro**(2TSOL), **shmget**(2TSOL), **shmop**(2TSOL)

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

NAME

shmget, shmgetl – Get shared-memory segment identifier

SYNOPSIS

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

int shmget(key_t *key***, size_t** *size***, int** *shmflg***);**

cc [*flag* **...]** *file* **... -ltsol [** *library* **...]**
#include <sys/tsol/ipcl.h>

int shmgetl(key_t *key***, size_t** *size***, int** *shmflg***, const bslabel_t** ∗*slabel***);**

DESCRIPTION

A shared-memory segment is identified by a unique combination of key and sensitivity label. This qualification of keys by sensitivity labels allows applications that use shared-memory segments to be run at multiple process sensitivity labels without inadvertently sharing data.

shmget( ) returns the shared-memory identifier associated with the union of *key* and the sensitivity label of the calling process.

shmgetl( ) returns the shared-memory identifier associated with the union of *key* and *slabel*. If the value of *slabel* does not match the sensitivity label of the calling process, then the effective privilege set of the process must include the necessary privileges: **PRIV_IPC_MAC_READ** and **PRIV_IPC_MAC_WRITE**.

If discretionary read / write access is denied to the calling process as speci- fied by the low-order 9 bits of *shmflg*, both **shmget** and **shmgetl** require one or both of these privileges: **PRIV_IPC_DAC_READ** and **PRIV_IPC_DAC_WRITE**.

A shared-memory identifier and associated data structure and shared-memory segment of at least *size* bytes [see **intro**(2)] are created for *key* if one of these conditions is true:

> *key* is equal to **IPC_PRIVATE**.
>
> *key* does not already have a shared-memory identifier associated with it, and (*shmflg***&IPC_CREAT**) is true.

Upon creation, the data structure associated with the new shared-memory identifier is initialized as follows:

> **shm_perm.cuid** and **shm_perm.uid** are set equal to the effective user ID, and **shm_perm.cgid**, and **shm_perm.gid** are set equal to the effective group IDs of the calling process.
>
> The access permission bits of **shm_perm.mode** are set equal to the access permission bits of *shmflg*. **shm_segsz** is set equal to the value of *size*.
>
> **shm_lpid**, **shm_nattch shm_atime**, and **shm_dtime** are set equal to **0**.
>
> **shm_ctime** is set equal to the current time.

If information labels are enabled, a new shared-memory segment is labeled ADMIN_LOW. An existing shared-memory segmentt remains unchanged.

RETURN VALUES

Upon successful completion, **shmget** and **shmgetl** return a nonnegative integer, namely, a shared-memory identifier, is returned. Upon failure, **shmget** and **shmgetl** return a value of −**1** and set **errno** to indicate the error.

ERRORS

shmget and **shmgetl** fail if any of these conditions is true:

EACCES A semaphore-structure identifier exists for the union of key and sensitivity label, but operation permission [see intro(2)] as specified by the low-order 9 bits of *semflg* would not be granted.

The calling process, which failed the check for discretionary or mandatory access, does not have the appropriate privilege override(s) in its set of effective privileges.

EEXIST A shared-memory identifier exists for *key*, but both (*shmflg*&**IPC_CREATE**) and (*shmflg*&**IPC_EXCL**) are true.

EFAULT *slabel* points to an illegal address.

EINVAL *size* is less than the system-imposed minimum or greater than the system-imposed maximum.

A shared-memory identifier exists for *key*, but the size of the segment associated with it is less than *size*, which is not equal to **0**.

The label pointed to by *slabel* is not a valid sensitivity label.

ENOENT A shared memory identifier does not exist for the union of key and sensitivity label, and (*shmflg*&**IPC_CREATE**) is false.

ENOMEM A shared-memory identifier and associated shared-memory segment are to be created, but the amount of available memory is not sufficient to fill the request.

ENOSPC A shared-memory identifier is to be created, but the system-imposed limit on the maximum number of allowed shared-memory identifiers systemwide would be exceeded.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

Sensitivity labels are used together with *key* to determine shared-memory identifiers.

SEE ALSO

intro(2TSOL), **shmctl**(2TSOL), **shmop**(2TSOL), **stdipc**(3C)

NOTES

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

NAME

shmop, shmat, shmdt – Shared-memory operations

SYNOPSIS

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

void ∗**shmat(int** *shmid,* **const void** ∗*shmaddr*, **int** *shmflg***);**

int shmdt(void ∗*shmaddr***);**

DESCRIPTION

shmat( ) attaches the shared-memory segment associated with the shared-memory identifier specified by *shmid* to the data segment of the calling process.

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

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

See the *Shared Memory Operation Permissions* section of **intro**(2TSOL) for more information.

A process attempting to map a shared-memory segment as read-only (*shmflg***&SHM_RDONLY**) must either have discretionary and mandatory read access to the shared-memory object or have the necessary privileges in its set of effective privileges: **PRIV_IPC_DAC_READ** and **PRIV_IPC_MAC_READ**. Otherwise, mapping the shared-memory segment for reading and writing requires that the process have discretionary and mandatory read access and discretionary and mandatory write access to the shared memory object, or that the effective privilege set of the process include these privileges as necessary: **PRIV_IPC_DAC_READ**, **PRIV_IPC_MAC_READ**, **PRIV_IPC_DAC_WRITE**, and **PRIV_IPC_MAC_WRITE**.

When (*shmflg***&SHM_SHARE_MMU**) is true, virtual-memory resources in addition to shared memory itself are shared among processes that use the same shared memory.

The shared-memory segment is attached to the data segment of the calling process at the address specified based on one of these criteria:

- If *shmaddr* is equal to **(void** ∗**) 0**, the segment is attached to the first available address as selected by the system.
- If *shmaddr* is equal to **(void** ∗**) 0** and (*shmflg***&SHM_SHARE_MMU**) is true, then the segment is attached to the first available aligned address. See **NOTES** for the alignment requirement.
- If *shmaddr* is not equal to **(void** ∗**) 0** and (*shmflg***&SHM_RND**) is true, the segment is attached to the address given by (*shmaddr* **-** (*shmaddr* modulus **SHMLBA**)).

- If *shmaddr* is not equal to **(void ∗) 0** and (*shmflg*&**SHM_RND**) is false, the segment is attached to the address given by *shmaddr*.

The segment is attached for reading if (*shmflg*&**SHM_RDONLY**) is true {READ}; otherwise, the segment is attached for reading and writing {READ / WRITE}.

When (*shmflg*&**SHM_SHARE_MMU**) is set, however, the permission given by **shmget** determines whether the segment is attached for reading or for reading and writing.

If information-label floating is enabled on this system, **shmat** may cause the information label of a process or of a shared-memory segment to float.

If the shared-memory segment is attached for reading only, the information label of the shared-memory segment is unchanged; and the process information label of the subject is floated by combining it with information label of the shared-memory segment unless the subject has asserted the **PRIV_PROC_NOFLOAT** privilege.

If the shared-memory segment allows writing, the information label of the shared-memory segment is floated by the information label of the subject unless the subject has asserted the **PRIV_IPC_NOFLOAT** privilege. Whenever the information label of the subject is floated, the information label of the shared-memory segment is also floated unless the subject has asserted the **PRIV_IPC_NOFLOAT** privilege. The information label of the subject will also be floated whenever the information label of the shared-memory segment is floated unless the subject has asserted the **PRIV_PROC_NOFLOAT** privilege. The process information label of the subject is floated by combining it with the information label of the shared-memory segment.

shmdt( ) detaches from the data segment of the calling process the shared-memory segment located at the address specified by *shmaddr*.

RETURN VALUES

Upon successful completion,: **shmat** returns the data-segment start address of the attached shared-memory segment; **shmdt** returns **0**. Upon failure, they return –**1** and set **errno** to indicate the error.

ERRORS

shmat fails and does not attach the shared-memory segment if any of these conditions is true:

EACCES　Operation permission is denied to the calling process [see intro(2)], and the calling process does not have the appropriate privilege(s) in its set of effective privileges.

EINVAL　*shmid* is not a valid shared-memory identifier.

shmaddr is not equal to zero, and the value of (*shmaddr* **-** (*shmaddr* modulus **SHMLBA**)) is an illegal address.

shmaddr is not equal to zero, (*shmflg*&**SHM_RND**) is false, and *shmaddr* is an illegal address.

shmaddr is not equal to zero, (*shmfg*&**SHM_SHARE_MMU**) is true, and *shmaddr* is not aligned properly.

shmdt( ) fails and does not detach the shared-memory segment if *shmaddr* is not the data-segment start address of a shared-memory segment.

EINVAL	**SHM_SHARE_MMU** is not supported in certain architectures.
	This operation would cause the information label of the shared-memory segment to dominate its sensitivity label, and the calling process does not have the appropriate privilege override(s) in its set of effective privileges.
	This operation would cause the information label of the calling process to float above its sensitivity label, and the calling process does not have the appropriate privilege override in its set of effective privileges.
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.
EPERM	The LOCK and UNLOCK operation does not have the appropriate privilege in its set of effective privileges.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

intro(2TSOL), **exec**(2TSOL), **exit**(2), **fork**(2TSOL), **shmctl**(2TSOL), **shmget**(2TSOL), **xpg4**(5)

NOTES

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

The alignment requirement, which varies on different machines, is determined by the mapping size of the virtual-memory system.

NAME

sigsend, sigsendset – send a signal to a process or a group of processes

SYNOPSIS

#include <signal.h>

int sigsend(idtype_t *idtype***, id_t** *id***, int** *sig***);**

int sigsendset(procset_t **psp***, int** *sig***);**

DESCRIPTION

sigsend( ) sends a signal to the process or group of processes specified by *id* and *idtype.* The signal to be sent is specified by *sig* and is either **0** or one of the values listed in **signal**(5). 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 sending process must have MAC write access to the receiving processes. The real or effective user ID of the sending process must match the real or saved user ID of the receiving process, unless the sending process has the **PRIV_PROC_OWNER** privilege, or *sig* is **SIGCONT** and the sending process has the same session ID as the receiving process.

If *idtype* is **P_PID**, *sig* is sent to the process with process ID *id.*

If *idtype* is **P_PGID**, *sig* is sent to any process with process group ID *id.*

If *idtype* is **P_SID**, *sig* is sent to any process with session ID *id.*

If *idtype* is **P_UID**, *sig* is sent to any process with effective user ID *id.*

If *idtype* is **P_GID**, *sig* is sent to any process with effective group ID *id.*

If *idtype* is **P_CID**, *sig* is sent to any process with scheduler class ID *id* (see **priocntl**(2TSOL)).

If *idtype* is **P_ALL**, *sig* is sent to all processes and *id* is ignored.

If *id* is **P_MYID**, the value of *id* is taken from the calling process.

The process with a process ID of 0 is always excluded. The process with a process ID of 1 is excluded unless *idtype* is equal to **P_PID**.

sigsendset( ) provides an alternate interface for sending signals to sets of processes. This function sends signals to the set of processes specified by *psp. psp* is a pointer to a structure of type **procset_t**, defined in **<sys/procset.h>**, which includes the following members:

```
idop_t      p_op;
idtype_t    p_lidtype;
id_t        p_lid;
idtype_t    p_ridtype;
id_t        p_rid;
```

p_lidtype and **p_lid** specify the ID type and ID of one (“left”) set of processes; **p_ridtype** and **p_rid** specify the ID type and ID of a second (“right”) set of processes. ID types and IDs are specified just as for the *idtype* and *id* arguments to **sigsend( )**. **p_op** specifies the operation to be performed on the two sets of processes to get the set of processes the function is to apply to. The valid values for **p_op** and the processes they specify are:

POP_DIFF	set difference: processes in left set and not in right set
POP_AND	set intersection: processes in both left and right sets
POP_OR	set union: processes in either left or right set or both
POP_XOR	set exclusive-or: processes in left or right set but not in both

RETURN VALUES

On success, **sigsend( )** returns **0**. On failure, it returns –**1** and sets **errno** to indicate the error.

ERRORS

sigsend( ) and **sigsendset( )** fail if one or more of the following are true:

EINVAL	*sig* is not a valid signal number.
EINVAL	*idtype* is not a valid idtype field.
EINVAL	*sig* is **SIGKILL**, *idtype* is **P_PID** and *id* is 1 (proc1).
EPERM	The calling process does not have the **PRIV_PROC_OWNER** privilege, and its real or effective user ID does not match the real or saved 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*. Or, the sending process does not have MAC write access to the specified process.

In addition, **sigsendset( )** fails if:

EFAULT	**psp** points to an illegal address.

SUMMARY OF TRUSTED SOLARIS CHANGES

The sending process is required to have MAC write access to the target processes. The **PRIV_PROC_MAC_WRITE** and **PRIV_PROC_OWNER** privileges are recognized.

SEE ALSO

kill(1), **getpid**(2TSOL), **kill**(2TSOL), **priocntl**(2TSOL), **signal**(3C), **signal**(5)

NAME

stat, lstat, fstat, mldstat, mldlstat – Get file status

SYNOPSIS

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

int stat(const char **path***, struct stat** **buf***);**

int lstat(const char **path***, struct stat** **buf***);**

int fstat(int *fildes***, struct stat** **buf***);**

int mldstat(const char **path***, struct stat** **buf***);**

int mldlstat(const char **path***, struct stat** **buf***);**

MT-LEVEL

stat( ), **fstat( )**, **mldstat( )**, and **mldlstat( )** are Async-Signal-Safe.

DESCRIPTION

stat( ) obtains information about the file to which *path* points. The calling process must have mandatory read access to *path*, and all directories listed in the path name must be searchable.

lstat( ) obtains file attributes similar to those that **stat** obtains except when the named file is a symbolic link; in that case, **lstat** returns information about the link; **stat** returns information about the file that the link references.

fstat( ) obtains information about an open file known by the file descriptor *fildes*, obtained from a successful **open**, **creat**, **dup**, **fcntl**, or **pipe** function.

buf is a pointer to a **stat** structure into which information concerning the file is placed.

The contents of the structure to which *buf* points include these members:

```
mode_t  st_mode;      /* File mode [See mknod(2TSOL).] */
ino_t      st_ino;         /* Inode number */
dev_t      st_dev;         /* ID of device containing */
                           /* a directory entry for this file */
dev_t      st_rdev;        /* ID of device */
                           /* This entry is defined only for */
                           /* char special or block special files */
nlink_t    st_nlink;       /* Number of links */
uid_t      st_uid;         /* User ID of the file's owner */
gid_t      st_gid;         /* Group ID of the file's group */
off_t      st_size;        /* File size in bytes */
time_t     st_atime;       /* Time of last access */
time_t     st_mtime;       /* Time of last data modification */
time_t     st_ctime;       /* Time of last file status change */
                           /* Times measured in seconds since */
                           /* 00:00:00 UTC, Jan. 1, 1970 */
long       st_blksize;     /* Preferred I/O block size */
long       st_blocks;      /* Number of 512 byte blocks allocated*/
```

st_mode	The mode of the file as described in **mknod**(2TSOL). In addition to the modes described in **mknod**(2TSOL), the mode of a file may also be **S_IFLNK** if the file is a symbolic link. [Note that **S_IFLNK** may be returned only by **lstat**.]
st_ino	This field uniquely identifies the file in a given file system. The pair **st_ino** and **st_dev** uniquely identifies regular files.
st_dev	This field uniquely identifies the file system that contains the file. The value of the field may be used as input to the **ustat( )** function to determine more information about this file system. No other meaning is associated with this value.
st_rdev	This field should be used only by administrative commands. It is valid only for block special or character special files and has meaning only on the system where the file was configured.
st_nlink	This field should be used only by administrative commands.
st_uid	The user ID of the file's owner
st_gid	The group ID of the file's group
st_size	For regular files, this is the address of the end of the file. For block special or character special, this is not defined. See also **pipe**(2).
st_atime	Time when file data was last accessed. Changed by these functions: **creat**, **mknod**, **pipe**, **utime**, and **read**.
st_mtime	Time when data was last modified. Changed by these functions: **creat**, **mknod**, **pipe**, **utime**, and **write**.
st_ctime	Time when file status was last changed. Changed by these functions: **chmod**, **chown**, **creat**, **link**, **mknod**, **pipe**, **unlink**, **utime**, and **write**.
st_blksize	A hint as to the "best" unit size for I/O operations. This field is not defined for block special or character special files.
st_blocks	The total number of physical blocks of size 512 bytes actually allocated on disk. This field is not defined for block special or character special files.

mldstat( ) obtains file attributes similar to those that **stat** obtains except when the referenced file is an SLD; in that case, **mldstat** returns information about the MLD; **stat** returns information about the underlying SLD.

mldlstat obtains file attributes similar to those that **lstat** obtains except when the named file is a MLD; in that case, **mldlstat** returns information about the MLD; **lstat** returns information about the underlying SLD.

stat, **lstat**, **fstat**, **mldstat**, and **mldlstat** require mandatory read access to the final component of *path* or the calling process must have the **PRIV_FILE_MAC_READ** privilege in its set of effective privileges. If the descriptor is open only for writing, **fstat** requires mandatory read access to the object to which the descriptor refers or the calling process must have the **PRIV_FILE_MAC_READ** privilege in its set of effective privileges. If the file being accessed is a pseudo-terminal (**/dev/ttyp**∗ or **/dev/ptyp**∗) and the calling process does not have mandatory read access or the calling process does not have **PRIV_FILE_MAC_READ** in its set of effective privileges, **stat**, **lstat**, and **fstat** return the values indicated for these elements of the stat structure:

```
mode_t  st_mode;         /* file mode - set to zero */
nlink_t st_nlink;        /* number of hard links - set to one */
uid_t   st_uid;          /* user ID of owner - set to zero */
gid_t   st_gid;          /* group ID of owner - set to zero */
time_t  st_atime;        /* last access time - set to current time */
time_t  st_mtime;        /* last modify time - set to current time */
time_t  st_ctime;/* last status change - set to current time*/
```

The information label of *path* and that of the calling process remain unchanged.

RETURN VALUES

Upon successful completion, these functions return a value of **0**. Upon failure, they return a value of −**1** and set **errno** to indicate the error.

ERRORS

stat and **lstat** fail if any of these conditions is true:

EACCES	Search permission is denied for a component of the path prefix. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_SEARCH** and **PRIV_FILE_MAC_SEARCH**.
EACCES	The calling process does not have mandatory read access to the final component of *path*. To override this restriction, the calling process may assert the **PRIV_FILE_MAC_READ** privilege.
EFAULT	*buf* or *path* points to an illegal address.
EINTR	A signal was caught during the **stat** or **lstat** function.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines but the file system does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	The named file does not exist or is a null path name.
ENOLINK	*path* points to a remote machine but 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 to which *buf* points.

fstat fails if any of these conditions is true:

EBADF	*fildes* is not a valid open file descriptor.
EFAULT	*buf* points to an illegal address.
EINTR	A signal was caught during the **fstat** function.
ENOLINK	*fildes* points to a remote machine but the link to that machine is no longer active.

EOVERFLOW	A component is too large to store in the structure to which *buf* points.
EIO	An I/O error occurred while reading from or writing to the file system.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

mldstat returns information about the MLD when the referenced file is an SLD.

mldlstat returns information about the MLD when the named file is an MLD.

The information label of *path* and that of the calling process remain unchanged.

Certain uses of this interface may present a covert channel. If a covert channel is exploited, the execution of the process may be delayed. To bypass this delay, the process may assert the **PRIV_PROC_NODELAY** privilege.

SEE ALSO

chmod(2TSOL), **chown**(2TSOL), **creat**(2TSOL), **link**(2TSOL), **mknod**(2TSOL), **pipe**(2), **read**(2TSOL), **time**(2), **unlink**(2TSOL), **utime**(2), **write**(2TSOL), **fattach**(3C), **stat**(5)

NAME

statvfs, fstatvfs – Get file system information

SYNOPSIS

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

int statvfs(const char **path***, struct statvfs** **buf***);**

int fstatvfs(int *fildes***, struct statvfs** **buf***);**

DESCRIPTION

statvfs( ) returns a “generic superblock” describing a file system; the function can be used to acquire information about mounted file systems. *buf* is a pointer to a structure (described hereafter) that is filled by the function.

path should name a file that resides on that file system. The file system type is known to the operating system. Discretionary 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 to which *buf* points includes these members:

```
u_long   f_bsize;                /* preferred file system block size */
u_long   f_frsize;               /* fundamental filesystem block size
                                 (if supported) */
u_long   f_blocks;               /* total # of blocks on file system
                                 in units of f_frsize */
u_long   f_bfree;                /* total # of free blocks */
u_long   f_bavail;               /* # of free blocks avail to
                                 non-super-user */
u_long   f_files;                /* total # of file nodes (inodes) */
u_long   f_ffree;                /* total # of free file nodes */
u_long   f_favail;               /* # of inodes avail to
                                 non-super-user*/
u_long   f_fsid;                 /* file system id (dev for now) */
char     f_basetype[FSTYPSZ];    /* target fs type name,
                                 null-terminated */
u_long   f_flag;                 /* bit mask of flags */
u_long   f_namemax;              /* maximum file name length */
char     f_fstr[32];             /* file system specific string */
u_long   f_filler[16];           /* reserved for future expansion */
```

f_basetype contains a null-terminated FSType name of the mounted target.

These flags can be returned in the **f_flag** field:

```
ST_RDONLY      0x01        /* read-only file system */
ST_NOSUID      0x02        /* does not support setuid/setgid
                           semantics */
ST_NOTRUNC     0x04        /* does not truncate file names
                           longer than {NAME_MAX}*/
```

fstatvfs( ) is similar to **statvfs** except that the file named by *path* in **statvfs** is instead identified by an open file descriptor *fildes* obtained from a successful **open**(2TSOL), **creat**(2TSOL), **dup**(2), **fcntl**(2TSOL), or **pipe**(2) function.

The information label of *path* or *fildes* and the information label of the calling process remain unchanged.

RETURN VALUES

Upon successful completion, these calls return a value of **0**. Upon failure, they return a value of –**1** and set **errno** to indicate the error.

ERRORS

statvfs fails if any of these conditions is true:

EACCES	Search permission is denied on a component of the path prefix. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_SEARCH** and **PRIV_FILE_MAC_SEARCH**.
	The calling process does not have mandatory read access to *path_name*. To override this restriction, the calling process may assert the **PRIV_FILE_MAC_READ** privilege.
EFAULT	*path* or *buf* points to an illegal address.
EINTR	A signal was caught during **statvfs** execution.
EIO	An I/O error occurred while reading the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines but the file system type does not allow it.
ENAMETOOLONG	The length of a *path* component exceeds {**NAME_MAX**} characters, or the length of *path* exceeds {**PATH_MAX**} characters.
ENOENT	Either a component of the path prefix or the file to which *path* refers does not exist.
ENOLINK	*path* points to a remote machine but the link to that machine is no longer active.
ENOTDIR	A component of the path prefix of *path* is not a directory.

fstatvfs fails if any of these conditions is true:

EACCES	The descriptor is open only for writing and the calling process does not have mandatory read access to the object to which the descriptor refers. To override this restriction, the calling process may assert the **PRIV_FILE_MAC_READ** privilege.
EBADF	*fildes* is not an open file descriptor.
EFAULT	*buf* points to an illegal address.
EINTR	A signal was caught during **fstatvfs** execution.
EIO	An I/O error occurred while reading the file system.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

The information label of *path* or *fildes* and the information label of the calling process remain unchanged.

SEE ALSO

chmod(2TSOL), **chown**(2TSOL), **creat**(2TSOL), **dup**(2), **fcntl**(2TSOL), **link**(2TSOL), **mknod**(2TSOL), **open**(2TSOL), **pipe**(2), **read**(2TSOL), **time**(2), **unlink**(2TSOL), **utime**(2), **write**(2TSOL)

BUGS

The values returned for **f_files**, **f_ffree**, and **f_favail** may not be valid for NFS-mounted file systems.

NAME

stime – set system time and date

SYNOPSIS

#include <unistd.h>

int stime(const time_t **tp***);**

DESCRIPTION

stime( ) sets the system's idea of the time and date. *tp* points to the value of time as measured in seconds from 00:00:00 UTC January 1, 1970. The calling process must have the **PRIV_SYS_CONFIG** privilege in order to use this system call.

RETURN VALUES

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

ERRORS

stime( ) will fail if:

EPERM	The calling process does not have the **PRIV_SYS_CONFIG** privilege.

SUMMARY OF TRUSTED SOLARIS CHANGES

The calling process must have the **PRIV_SYS_CONFIG** privilege in order to use this system call.

SEE ALSO

time(2)

NAME

swapctl – manage swap space

SYNOPSIS

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

int swapctl(int cmd, void *arg);
```

DESCRIPTION

swapctl( ) adds, deletes, or returns information about swap resources. *cmd* specifies one of the following options contained in **<sys/swap.h>**:

```
SC_ADD              /* add a resource for swapping */
SC_LIST             /* list the resources for swapping */
SC_REMOVE           /* remove a resource for swapping */
SC_GETNSWP          /* return number of swap resources */
```

When **SC_ADD** or **SC_REMOVE** is specified, *arg* is a pointer to a **swapres** structure containing the following members:

```
char      *sr_name;             /* pathname of resource */
off_t     sr_start;             /* offset to start of swap area */
off_t     sr_length;            /* length of swap area */
```

sr_start and **sr_length** are specified in 512-byte blocks.

When **SC_LIST** is specified, *arg* is a pointer to a **swaptable** structure containing the following members:

```
int       swt_n;                /* number of swapents following */
struct    swapent swt_ent[];    /* array of swt_n swapents */
```

A **swapent** structure contains the following members:

```
char      *ste_path;            /* name of the swap file */
off_t     ste_start;            /* starting block for swapping */
off_t     ste_length;           /* length of swap area */
long      ste_pages;            /* number of pages for swapping */
long      ste_free;             /* number of ste_pages free */
long      ste_flags;            /* ST_INDEL bit set if swap file */
                                /* is now being deleted */
```

SC_LIST causes **swapctl( )** to return at most **swt_n** entries. The return value of **swapctl( )** is the number actually returned. The **ST_INDEL** bit is turned on in **ste_flags** if the swap file is in the process of being deleted.

When **SC_GETNSWP** is specified, **swapctl( )** returns as its value the number of swap resources in use. *arg* is ignored for this operation.

The **SC_ADD** and **SC_REMOVE** functions will fail if calling process does not have appropriate privileges.

RETURN VALUES

Upon successful completion, the function **swapctl( )** returns a value of **0** for **SC_ADD** or **SC_REMOVE**, the number of **struct swapent** entries actually returned for **SC_LIST**, or the number of swap resources in use for **SC_GETNSWP**. Upon failure, the function **swapctl( )** returns a value of –**1** and sets **errno** to indicate an error.

ERRORS

Under the following conditions, the function **swapctl( )** fails and sets **errno** to:

EEXIST	Part of the range specified by **sr_start** and **sr_length** is already being used for swapping on the specified resource (**SC_ADD**).
EFAULT	*arg*, **sr_name**, or **ste_path** points to an illegal address.
EINVAL	The specified function value is not valid, the path specified is not a swap resource (**SC_REMOVE**), part of the range specified by **sr_start** and **sr_length** lies outside the resource specified (**SC_ADD**), or the specified swap area is less than one page (**SC_ADD**).
EISDIR	The path specified for **SC_ADD** is a directory.
ELOOP	Too many symbolic links were encountered in translating the pathname provided to **SC_ADD** or **SC_REMOVE** .
ENAMETOOLONG	The length of a component of the path specified for **SC_ADD** or **SC_REMOVE** exceeds **{NAME_MAX}** characters or the length of the path exceeds **{PATH_MAX}** characters and **{_POSIX_NO_TRUNC}** is in effect.
ENOENT	The pathname specified for **SC_ADD** or **SC_REMOVE** does not exist.
ENOMEM	An insufficient number of **struct swapent** structures were provided to **SC_LIST**, or there were insufficient system storage resources available during an **SC_ADD** or **SC_REMOVE**, or the system would not have enough swap space after an **SC_REMOVE**.
ENOSYS	The pathname specified for **SC_ADD** or **SC_REMOVE** is not a file or block special device.
ENOTDIR	Pathname provided to **SC_ADD** or **SC_REMOVE** contained a component in the path prefix that was not a directory.
EPERM	The effective user of the calling process is not super-user. To override this restriction, the calling process must assert the PRIV_SYS_MOUNT privilege.
EROFS	The pathname specified for **SC_ADD** is a read-only file system.

SUMMARY OF TRUSTED SOLARIS CHANGES

For a successful call, the calling process must assert the PRIV_SYS_MOUNT privilege.

NAME

symlink – Make a symbolic link to a file

SYNOPSIS

#include <unistd.h>

int symlink(const char **name1***, const char** **name2***);**

DESCRIPTION

symlink( ) creates a symbolic link *name2* to the file *name1*. Either name may be an arbitrary path name, 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**(2TSOL) operation is performed on the link. A **stat**(2TSOL) on a symbolic link returns the linked-to file; an **lstat** returns information about the link itself. This difference can lead to surprising results when a symbolic link is made to a directory. To avoid confusion in programs, the **readlink**(2TSOL) call can be used to read the contents of a symbolic link.

The containing directory cannot be a multilevel directory. There is no privilege to bypass this restriction.

NAME

sysinfo – get and set system information strings

SYNOPSIS

#include <sys/systeminfo.h>

long sysinfo(int *command*, **char** **buf*, **long** *count***);**

DESCRIPTION

sysinfo( ) copies information relating to the operating system on which the process is executing into the buffer pointed to by *buf*. **sysinfo( )** can also set certain information where appropriate *commands* are available. *count* is the size of the buffer.

The POSIX P1003.1 interface **sysconf**(3C) provides a similar class of configuration information, but returns an integer rather than a string.

The *command*s available are:

SI_SYSNAME Copy into the array pointed to by *buf* the string that would be returned by **uname**(2) in the *sysname* field. This is the name of the implementation of the operating system, for example, **SunOS** or **UTS**.

SI_HOSTNAME Copy into the array pointed to by *buf* a string that names the present host machine. This is the string that would be returned by **uname**(2) in the *nodename* field. This hostname or nodename is often the name the machine is known by locally.

The *hostname* is the name of this machine as a node in some network. Different networks may have different names for the node, but presenting the nodename to the appropriate network directory or name-to-address mapping service should produce a transport end point address. The name may not be fully qualified.

Internet host names may be up to **256** bytes in length (plus the terminating null).

SI_SET_HOSTNAME Copy the null-terminated contents of the array pointed to by *buf* into the string maintained by the kernel whose value will be returned by succeeding calls to **sysinfo( )** with the command **SI_HOSTNAME**. This command requires that the calling process have the **PRIV_SYS_NET_CONFIG** privilege.

SI_RELEASE Copy into the array pointed to by *buf* the string that would be returned by **uname**(2) in the *release* field. Typical values might be **5.2** or **4.1**.

SI_VERSION Copy into the array pointed to by *buf* the string that would be returned by **uname**(2) in the *version* field. The syntax and semantics of this string are defined by the system provider.

SI_MACHINE Copy into the array pointed to by *buf* the string that would be returned by **uname**(2) in the *machine* field, for example, **sun4c**, **sun4d**, or **sun4m**.

SI_ARCHITECTURE	Copy into the array pointed to by *buf* a string describing the instruction set architecture of the current system, for example, **sparc**, **mc68030**, **m32100**, or **i386**. These names may not match predefined names in the C language compilation system.
SI_PLATFORM	Copy into the array pointed to by *buf* a string describing the specific model of the hardware platform, for example, **SUNW,Sun_4_75**, **SUNW,SPARCsystem-600**, or **i86pc**.
SI_HW_PROVIDER	Copies the name of the hardware manufacturer into the array pointed to by *buf*.
SI_HW_SERIAL	Copy into the array pointed to by *buf* a string which is the ASCII representation of the hardware-specific serial number of the physical machine on which the function is executed. Note that this may be implemented in Read-Only Memory, using software constants set when building the operating system, or by other means, and may contain non-numeric characters. It is anticipated that manufacturers will not issue the same "serial number" to more than one physical machine. The pair of strings returned by **SI_HW_PROVIDER** and **SI_HW_SERIAL** is likely to be unique across all vendor's SVR4 implementations.
SI_SRPC_DOMAIN	Copies the Secure Remote Procedure Call domain name into the array pointed to by *buf*.
SI_SET_SRPC_DOMAIN	Set the string to be returned by **sysinfo( )** with the **SI_SRPC_DOMAIN** command to the value contained in the array pointed to by *buf*. This command requires that the calling process have the **PRIV_SYS_NET_CONFIG** privilege.

RETURN VALUES

Upon successful completion, the value returned indicates the buffer size in bytes required to hold the complete value and the terminating null character. If this value is no greater than the value passed in *count*, the entire string was copied. If this value is greater than *count*, the string copied into *buf* has been truncated to *count –1* bytes plus a terminating null character.

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

ERRORS

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

EFAULT	*buf* does not point to a valid address.
EINVAL	The data for a SET command exceeds the limits established by the implementation.
EPERM	The calling process does not have the **PRIV_SYS_NET_CONFIG** privilege.

USAGE

In many cases there is no corresponding programmatic interface to set these values; such strings are typically settable only by the system administrator modifying entries in the **/etc/system** directory or the code provided by the particular OEM reading a serial number or code out of read-only memory, or hard-coded in the version of the operating system.

A good starting guess for *count* is 257, which is likely to cover all strings returned by this interface in typical installations.

SUMMARY OF TRUSTED SOLARIS CHANGES

The calling process must have the **PRIV_SYS_NET_CONFIG** privilege in order to perform the SE_SET_HOSTNAME, SI_SET_KERB_REALM, and SI_SET_SRPC_DOMAIN operations.

SEE ALSO

uname(2), **gethostid**(3C), **gethostname**(3C), **sysconf**(3C)

NAME

tokmapper – Manipulate kernel token mapping caches

SYNOPSIS

#include <netinet/in.h>
#include <sys/tiuser.h>
#include <sys/tsol/tndb.h>
int tokmapper(int *cmd***, void** **buf***)**

AVAILABILITY

SUNWtsolu

DESCRIPTION

tokmapper( ) manipulates kernel token mapping caches. *cmd* is the operation to be performed. Currently, the only operation supported is **MSIX_FLUSH**, which flushes kernel token mappings for the specified MSIX host. For the **MSIX_FLUSH** operation, *buf* should point to a **netbuf** structure declared in **<sys/tiuser.h>**. The network address in the **netbuf** structure should be a **sockaddr_in** structure declared in **<netinet/in.h>**.

To make this call successfully, a process must have the **PRIV_SYS_NET_CONFIG** privilege.

RETURN VALUES

tokmapper returns

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

ERRORS

EFAULT	*buf* points to an invalid address.
EINVAL	A field in the **netbuf** or **sockaddr_in** structure is invalid; or the operation specified in *cmd* is not supported.
EPERM	The process has insufficient privilege to perform the operation. To make this call successfully, a process must have the **PRIV_SYS_NET_CONFIG** privilege.

SEE ALSO

tokmapd(1MTSOL),
tokmapctl(1MTSOL)

NAME

uadmin – administrative control

SYNOPSIS

#include <sys/uadmin.h>

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

DESCRIPTION

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

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

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

AD_HALT	Halt the processor(s).
AD_POWEROFF	Halt the processor(s) and turn off the power.
AD_BOOT	Reboot the system, using the kernel file.
AD_IBOOT	Interactive reboot; user is prompted for bootable program name.

The calling process must have the **PRIV_SYS_BOOT** privilege in order to perform this command.

A_REBOOT The system stops immediately without any further processing. The action to be taken next is specified by *fcn* as above. The calling process must have the **PRIV_SYS_BOOT** privilege in order to perform this command.

A_REMOUNT The root file system is mounted again after having been fixed. This should be used only during the startup process. The calling process must have the **PRIV_SYS_MOUNT** privilege in order to perform this command.

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.

The calling process must have the **PRIV_SYS_BOOT** privilege in order to perform this command.

RETURN VALUES

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

A_SHUTDOWN	Never returns.
A_REBOOT	Never returns.
A_FREEZE	**0** upon resume.
A_REMOUNT	**0**

Upon unsuccessful completion, **–1** is returned and **errno** is set to indicate the error.

ERRORS

uadmin( ) fails if any of the following are true:

EPERM	The calling process does not have sufficient privilege.
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.

SUMMARY OF TRUSTED SOLARIS CHANGES

The calling process must have the **PRIV_SYS_BOOT** privilege in order to perform the A_FREEZE, A_REBOOT, and A_SHUTDOWN command. The calling process must have the **PRIV_SYS_MOUNT** privilege in order to perform the A_REMOUNT command.

SEE ALSO

kernel(1M), **uadmin**(1MTSOL)

NAME

ulimit – get and set process limits

SYNOPSIS

#include <ulimit.h>

long ulimit(int *cmd***, /*** *newlimit* ***/ . . .);**

DESCRIPTION

This function provides for control over process limits. The *cmd* values available are:

UL_GETFSIZE	Get the regular file size limit of the process. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.
UL_SETFSIZE	Set the regular file size limit of the process to the value of *newlimit* , taken as a **long**. Any process may decrease this limit, but only a process with an effective **PRIV_SYS_CONFIG** privilege may increase the limit.
UL_GMEMLIM	Get the maximum possible break value (see **brk**(2)).
UL_GDESLIM	Get the current value of the maximum number of open files per process configured in the system.

The **getrlimit( )** and **setrlimit( )** functions provide a more general interface for controlling process limits.

RETURN VALUES

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

ERRORS

ulimit( ) fails if the following is true:

EINVAL	The *cmd* argument is not valid.
EPERM	The process does not have **PRIV_SYS_CONFIG** effective privilege.

SUMMARY OF TRUSTED SOLARIS CHANGES

The **PRIV_SYS_CONFIG** privilege is checked where super-user ID is checked in base Solaris.

SEE ALSO

brk(2), **getrlimit**(2TSOL), **write**(2TSOL)

NOTES

ulimit( ) is effective in limiting the growth of regular files. Pipes are limited to {**PIPE_MAX**} bytes.

NAME

umount – Unmount a file system

SYNOPSIS

#include <sys/mount.h>

int umount(const char **file***);**

DESCRIPTION

umount( ) requests that a previously mounted file system contained on the block special device or directory identified by *file* be unmounted. *file* is a pointer to a path name. After the file system is unmounted, the directory upon which the file system was mounted reverts to its ordinary interpretation.

For all file system types except *namefs*, **umount** may be invoked by a calling process with the **PRIV_SYS_MOUNT** privilege. For the *namefs* file system, the calling process must either be the owner of *file* or assert the **PRIV_FILE_OWNER** privilege.

RETURN VALUES

Upon successful completion, **umount** returns a value of **0**. Upon failure, **umount** returns a value of –**1** and sets **errno** to indicate the error.

ERRORS

umount will fail if any of these conditions is true:

EACCES	Search permission is denied on a component of *file*. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_SEARCH** and **PRIV_FILE_MAC_SEARCH**.
EBUSY	A file on *file* is busy.
EFAULT	*file* points to an illegal address.
EINVAL	*file* is not mounted.
ENOENT	*file* does not exist.
ELOOP	Too many symbolic links were encountered in translating the path to which *file* points.
EMULTIHOP	Components of the path to which *file* points require hopping to multiple remote machines.
ENAMETOOLONG	The length of the *file* argument exceeds {**PATH_MAX**}, or the length of a *file* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOLINK	*file* is on a remote machine, but the link to that machine is no longer active.
ENOTBLK	*file* is not a block special device.
EPERM	The calling process does not own *file* and *file* is a file system of type *namefs*. To override this restriction, the calling process may assert the **PRIV_FILE_OWNER** privilege. *file* is not a file system of type *namefs* and the calling process has not asserted the **PRIV_SYS_MOUNT** privilege.

EREMOTE	*file* is remote.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access or ownership checks.

The information label of *file* and that of the calling process remain unchanged.

For all file system types except *namefs*, the **umount** system call may be invoked by a calling process with the **PRIV_SYS_MOUNT** privilege. For the *namefs* file system, the calling process must either be the owner of *file* or assert the **PRIV_FILE_OWNER** privilege.

SEE ALSO

mount(2TSOL)

NAME

unlink – Remove directory entry

SYNOPSIS

#include <unistd.h>

int unlink(const char **path***);**

MT-LEVEL

Async-Signal-Safe

DESCRIPTION

unlink( ) removes the directory entry to which *path* points and decrements the link count of the file referenced by the directory entry. When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, space occupied by the file is not released until all references to the file have been closed. If *path* is a symbolic link, the symbolic link is removed. *path* should not name a directory unless the process has asserted the **PRIV_SYS_CONFIG** privilege. Applications should use **rmdir** to remove directories.

Upon successful completion, **unlink** marks for update the **st_ctime** and **st_mtime** fields of the parent directory. Also, if the file's link count is not zero, the **st_ctime** field of the file is marked for update.

The information label of the directory or file is unchanged. The information label of the calling process is also unchanged.

RETURN VALUES

Upon successful completion, **unlink** returns **0**. Upon failure, **unlink** returns –**1** and sets **errno** to indicate the error.

ERRORS

The named file is unlinked unless any of these conditions is true:

EACCES Search permission is denied for a component of the *path* prefix. To override this restriction, the calling process must assert one or both of these privileges: **PRIV_FILE_DAC_SEARCH** and **PRIV_FILE_MAC_SEARCH**.

EACCES Write permission is denied on the directory containing the link to be removed. To override this restriction, the calling process must assert one or both of these privileges: **PRIV_FILE_DAC_WRITE** and **PRIV_FILE_MAC_WRITE**.

The calling process needs both mandatory read and write access to *path* and does not have the combination. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_MAC_READ** and **PRIV_FILE_MAC_WRITE**.

The parent directory has the sticky bit set and the file is not writable by the user; the user does not own the parent directory and the user does not own the file. To override this restriction, the calling process must assert one or more of these privileges: **PRIV_FILE_DAC_WRITE**, **PRIV_FILE_MAC_WRITE**, and **PRIV_FILE_OWNER**.

EBUSY	The entry to be unlinked is the mount point for a mounted file system.
EFAULT	*path* points to an illegal address.
EINTR	A signal was caught during the **unlink** function.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMULTIHOP	Components of *path* require hopping to multiple remote machines and the file system does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds {**PATH_MAX**}, or the length of a *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	The named file does not exist or is a null path name.
ENOLINK	*path* points to a remote machine but the link to that machine is no longer active.
ENOTDIR	A component of the *path* prefix is not a directory.
EPERM	If *path* is a directory, the calling process must assert the **PRIV_SYS_CONFIG** privilege.
EROFS	The directory entry to be unlinked is part of a read-only file system.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

If the named file is a directory, the calling process must assert the **PRIV_SYS_CONFIG** privilege.

The information label of the directory or file is not changed by this system call. If removing a file or directory from its containing directory causes the containing directory to be empty, the information label of the containing directory is set to ADMIN_LOW. The information label of the calling process is unchanged.

SEE ALSO

rm(1), **close**(2), **link**(2TSOL), **open**(2TSOL), **rmdir**(2TSOL)

NAME

utimes – Set file times

SYNOPSIS

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

int utimes(char ∗*file***, struct timeval** ∗*tvp***);**

DESCRIPTION

utimes( ) sets the access and modification times of the file named by *file*.

If *tvp* is **NULL**, the access and modification times are set to the current time. A process must be the owner of the file or must assert the **PRIV_FILE_OWNER** privilege to use **utimes** in this manner.

If it is not **NULL**, *tvp* is assumed to point to an array of two **timeval** structures. The access time is set to the value of the first member, and the modification time is set to the value of the second member. Only the owner of the file or a process asserting the **PRIV_FILE_OWNER** privilege may use **utimes** in this manner.

In either case, the *inode-changed* time of the file is set to the current time.

utimes also causes the time of the last file status change (**st_ctime**) to be updated. The information labels of *file* and of the process remain unchanged.

RETURN VALUES

Upon success, **utimes** returns **0**. Upon failure, **utimes** returns –**1** and sets **errno** to indicate the error.

ERRORS

utimes( ) fails if any of these conditions is true:

EACCES	Search permission is denied for a component of the path prefix of *file*. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_SEARCH** and **PRIV_FILE_MAC_SEARCH**.
EACCES	Write permission is denied to the final component of *file*. To override this restriction, the calling process may assert one or both of these privileges: **PRIV_FILE_DAC_WRITE** and **PRIV_FILE_MAC_WRITE**.
EFAULT	*file* or *tvp* points to an illegal address.
EINTR	A signal was caught during the **utimes** function.
EINVAL	The number of microseconds specified in one or both of the **timeval** structures to which *tvp* points was greater than or equal to 1,000,000 or less than 0.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *file*.
EMULTIHOP	Components of *file* require hopping to multiple remote machines but the file system does not allow it.

ENAMETOOLONG	The length of the *file* argument exceeds {**PATH_MAX**}, or the length of a path component of *file* exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.
ENOENT	The named file does not exist or is a null path name.
ENOLINK	*file* points to a remote machine but the link to that machine is no longer active.
ENOTDIR	A component of the path prefix of *file* is not a directory.
EPERM	The effective user ID does not own the file, and *tvp* is not **NULL**. To override this restriction, the calling process may assert the **PRIV_FILE_OWNER** privilege.
EROFS	The file system containing the file is mounted read-only.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

The information label of *path* and that of the calling process remain unchanged.

To change the access and modification times on a file not owned by the calling process, the calling process may assert the **PRIV_FILE_OWNER** privilege.

SEE ALSO

stat(2TSOL)

NAME

vfork – Spawn new process in a virtual-memory-efficient way

SYNOPSIS

#include <unistd.h>

pid_t vfork(void);

DESCRIPTION

vfork( ) can be used to create new processes without fully copying the address space of the old process. **vfork** is useful when the purpose of **fork** would have been to create a new system context for an **execve( )**. **vfork** differs from **fork** in that the child borrows the parent's memory and thread of control until a call to **execve( )** or an exit either normally by a call to **_exit( )** [see **exit**(2)] or abnormally. The parent process is suspended while the child is using its resources.

vfork returns **0** in the child's context and (later) the process ID (PID) of the child in the parent's context.

vfork can normally be used just like **fork**. **vfork** does not work, however, to return while running in the child's context from the procedure that called **vfork** because the eventual return from **vfork** would then return to a no-longer-existent stack frame. Be careful also to call **_exit( )** rather than **exit**(3C) if you cannot **execve( )** because **exit**(3C) will flush and close standard I/O channels, and thereby corrupt the parent processes standard I/O data structures. Even with **fork**, it is wrong to call **exit**(3C) because buffered data would then be flushed twice.

RETURN VALUES

Upon successful completion, **vfork** returns a value of **0** to the child process and returns the process ID of the child process to the parent process. Upon failure, **vfork** returns a value of –**1** to the parent process, no child process is created, and **vfork** sets the global variable **errno** to indicate the error.

ERRORS

vfork will fail and no child process will be created if any of these conditions is true:

EAGAIN The system-imposed limit on the total number of processes under execution would be exceeded. Moreover, the calling process does not have the **PRIV_SYS_MAXPROC** privilege to override the limit. This limit is determined when the system is generated.

The system-imposed limit on the total number of processes under execution by a single user would be exceeded. Moreover, the calling process does not have the **PRIV_SYS_MAXPROC** privilege to override the limit. This limit is determined when the system is generated.

ENOMEM There is insufficient swap space for the new process.

SUMMARY OF TRUSTED SOLARIS CHANGES

Before a child process exits or executes a new program, thus no longer sharing memory with the parent, the information label of the parent process will float up whenever an information label float event occurred to the child. A process may override this floating by asserting the **PRIV_PROC_NOFLOAT** privilege. A **PRIV_PROC_NOFLOAT** privilege in a child process will not override parent's floating.

A process with the **PRIV_SYS_MAXPROC** privilege may override the limit on the number of processes a user may have.

SEE ALSO

exec(2TSOL), **exit**(2), **fork**(2TSOL), **ioctl**(2), **wait**(2), **exit**(3C)

NOTES

vfork is unsafe in multithread applications.

This function will be eliminated in a future release. The memory-sharing semantics of **vfork** can be obtained through other mechanisms.

To avoid a possible deadlock situation, processes that are children in the middle of a **vfork** are never sent **SIGTTOU** or **SIGTTIN** signals; rather, output or ioctls are allowed, and input attempts result in an EOF indication.

On some systems, the implementation of **vfork** causes the parent to inherit register values from the child. This can create problems for certain optimizing compilers if **<unistd.h>** is not included in the source calling **vfork**.

NAME

write, writel, pwrite, pwritel, writev, writevl – write on a file

SYNOPSIS

#include <unistd.h>

ssize_t write(int *fildes*, **const void** *∗buf*, **size_t** *nbyte***);**

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

ssize_t pwrite(int *fildes*, **const void** *∗buf*, **size_t** *nbyte*, **off_t** *offset***);**

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

int writev(int *fildes*, **const struct iovec** *∗iov*, **int** *iovcnt***);**

#include <tsol/rdwrl.h>

ssize_t writel(int *fildes*, **void** *∗buf*, **size_t** *nbyte*, **bclabel_t** *∗label_p***);**

ssize_t pwritel(int *fildes*, **void** *∗buf*, **size_t** *nbyte*, **off_t** *offset*, **bclabel_t** *∗label_p***);**

ssize_t writevl(int *fildes*, **struct iovec** *∗iov*, **int** *iovcnt*, **bclabel_t** *∗label_p***);**

MT-LEVEL

write( ) is Async-Signal-Safe.

DESCRIPTION

attempts to write *nbyte* bytes from the buffer to which *buf* points to the file descriptor specified by *fildes*. If *nbyte* is zero and the file is a regular file, **write** returns zero and has no other results.

pwrite( ) performs the same action as **write**, except that **pwrite** writes into a given position without changing the file pointer. The first three arguments to **pwrite** are the same as those for **write** with the addition of a fourth argument *offset* for the desired position inside the file.

writev( ) performs the same action as **write** but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], . . ., *iov*[*iovcnt* – 1]. The *iovcnt* buffer is valid if greater than 0 and less than or equal to **{IOV_MAX}**. [See #ifdef tSOL **intro**(2TSOL) #else **intro**(2) #endif for a definition of **{IOV_MAX}**.]

The **iovec** structure contains these members:

```
caddr_t    iov_base;
int        iov_len;
```

Each **iovec** entry specifies the base address and length of an area in memory from which data should be written. **writev** always writes all data from one area before proceeding to the next.

On devices capable of seeking, the actual writing of data starts at the position in the file indicated by the file pointer. On return from **write**, the file pointer is incremented by the number of bytes actually written. On a regular file, if the incremented file pointer is greater than the length of the file, the length of the file is set to the new file pointer.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the **O_APPEND** flag of the file status flags is set, the file pointer is set to the end of the file prior to each **write**. The system guarantees that no intervening file-modification operation will occur between changing the file offset and the write operation.

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

A **write** to a regular file is blocked if mandatory file / record locking is set [see **chmod**(2TSOL)] and a record lock owned by another process exists 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 until 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**(2TSOL) and **ulimit**(2TSOL)], 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 nonzero number of bytes gives a failure return (except as noted for pipes and FIFO hereafter).

Write requests to a pipe or FIFO are handled the same as those to a regular file with these exceptions:

- If there is no file offset associated with a pipe, 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** flag is 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, **write** returns –**1** and sets **errno** to **EAGAIN**; or if **O_NDELAY** is set, **write** returns **0**. A write request for greater than **{PIPE_BUF}** bytes transfers what it can and returns the number of bytes written; or the request transfers no data and, if **O_NONBLOCK** is set, returns –**1** with **errno** set to **EAGAIN**; or if **O_NDELAY** is set, the request returns **0**. Finally, if a request is greater than **{PIPE_BUF}** bytes and all data previously written to the pipe has been read, **write** transfers at least **{PIPE_BUF}** bytes.

When attempting to write to a file descriptor (other than a pipe, FIFO, or stream) that supports nonblocking writes and cannot accept the data immediately,

- If **O_NONBLOCK** and **O_NDELAY** are clear, **write** blocks until the data can be accepted.
- If **O_NONBLOCK** or **O_NDELAY** is set, **write** does not block the process. If some data can be written without blocking the process, **write** writes what it can and returns the number of bytes written. Otherwise, if **O_NONBLOCK** is set, **write** returns –**1** and sets **errno** to **EAGAIN**; or if **O_NDELAY** is set, **write** returns **0**.

For STREAMS files [see **intro**(2TSOL) 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 cannot 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 nonzero, **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).]

During writing to a stream, data messages are created with a priority band of zero. During writing to a stream that is not a pipe or FIFO,

- If **O_NDELAY** and **O_NONBLOCK** are not set, and the stream cannot accept data (the stream write queue is full because of 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.

writel( ), **pwritel**( ), and **writevl**( ) perform the same actions as **write**, **pwrite**, and **writev**, respectively, and additionally provide the CMW label *label_p* to associate with the data that is written. The label associated with the data that is written to *fd* depends on these conditions:

- If the descriptor refers to a file or a FIFO, then the sensitivity label portion of *label_p* is ignored.
- If the descriptor refers to a regular file, then the information label portion is conjoined with the current information label of the file, and this resulting information label becomes the new information label of the file.
- If the descriptor refers to a FIFO, the information label portion is associated with the data that is being written instead of with the information label of the calling process.

If the information label of the calling process is equal to the information label portion of *label_p*, then neither the **PRIV_FILE_UPGRADE_IL** nor **PRIV_FILE_DOWNGRADE_IL** privilege is required. If the information label portion of *label_p* dominates but is not equal to the information label of the calling process (an upgrade), then the calling process must assert **PRIV_FILE_UPGRADE_IL** in its set of effective privileges. If the information label portion of *label_p* does not dominate the information label of the calling process (a downgrade), then the calling process must assert **PRIV_FILE_DOWNGRADE_IL** in its set of effective privileges.

In all other respects, the **writel**, **pwritel**, and **writevl** interfaces are analogous to the **write**, **pwrite**, and **writev** interfaces.

For the **write**, **writel**, **pwrite**, **pwritel**, **writev**, and **writevl** interfaces, if the descriptor refers to a regular file and the **PRIV_FILE_NOFLOAT** privilege is contained in the set of effective privileges of the calling process, then the information label of the file is not floated. Otherwise, the information label associated with the data that is being written is conjoined with the current information label of the file, and this resulting information label becomes the new information label of the file.

If the descriptor refers to a FIFO, the information label associated with the data that is being written is associated with the new quantum of data. If the calling process asserts the **PRIV_FILE_NO_FLOAT** privilege, the information label of the new quantum of data is set to ADMIN_LOW.

If the set-user-ID or get-group-ID bits of *fildes* are set, they are cleared by the write. The calling process may assert the **PRIV_FILE_SETID** privilege to suppress this action.

If the forced or allowed privilege sets of *fildes* are not empty, they are cleared by the write. The calling process may assert the **PRIV_FILE_SETPRIV** privilege to suppress this action.

If the public object attributes flag, FAF_PUBLIC, of *fildes* is set, the flag is cleared by the write. The calling process may assert the **PRIV_FILE_AUDIT** privilege to suppress this action.

If the write causes the file-system free space to fall below its minimum level, the write fails. The calling process may assert the **PRIV_SYS_MINFREE** privilege to bypass this restriction.

When the calling process writes to a regular file, the process information label floats the information label of the file unless the **PRIV_FILE_NOFLOAT** privilege is asserted. For conduits (FIFOs, pipes, pseudo terminals, and STREAMS), the information label of the calling process is associated with each byte of data in the conduit.

The information label of the calling process is unchanged.

RETURN VALUES

On success, **write** returns the number of bytes actually written. Otherwise, **write** returns **-1** and sets **errno** to indicate the error.

ERRORS

write, **writel**, **pwrite**, **pwritel**, **writev**, and **writevl** fail and the file pointer remains unchanged if any of these conditions is true:

EAGAIN Mandatory file / record locking is set, **O_NDELAY** or **O_NONBLOCK** is set, and there is a blocking record lock.

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

An attempt is made to write to a stream that cannot accept data with the **O_NDELAY** or **O_NONBLOCK** flag set.

A write of **{PIPE_BUF}** bytes or less is requested to a pipe or FIFO, and less than *nbytes* of free space is available.

EBADF	*fildes* is not a valid file descriptor open for writing.
EDEADLK	The write was ready 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	*buf* points to an illegal address.
EFBIG	An attempt to write a file exceeds the file-size limit of the process or the maximum file size. [See **getrlimit**(2TSOL) and **ulimit**(2TSOL).]
EINTR	A signal was caught during the write operation and no data was transferred.
EINVAL	An attempt is made to write to a stream linked below a multiplexor.
EIO	The process in the background is attempting to write to its controlling terminal whose **TOSTOP** flag is set; the process is neither ignoring nor blocking **SIGTTOU** signals, and the process group of the process is orphaned.
ENOLCK	Enforced record locking was enabled and **{LOCK_MAX}** regions are already locked in the system. The system record-lock table was full, so the write could not go to sleep until the blocking record lock was removed.
ENOLINK	*fildes* is on a remote machine but the link to that machine is no longer active.
ENOSPC	During a write to an ordinary file, there is no free space left on the device.
ENOSR	An attempt is made to write to a stream with insufficient STREAMS memory resources available in the system.
ENXIO	A hangup occurred on the stream being written to.
EPIPE and **SIGPIPE** signal	An attempt is made to write either to a pipe that is not open for reading by any process or to a file descriptor created by **socket**(3N), using type **SOCK_STREAM** that is no longer connected to a peer endpoint. An attempted write of this kind also causes you to receive a **SIGPIPE** signal from the kernel. If you have not made special provision to catch or ignore this signal, your process dies.
EPIPE	An attempt is made to write to a FIFO that is not open for reading by any process. An attempt is made to write to a pipe that has only one end open.

ERANGE An attempt is made to write to a stream with *nbyte* outside specified minimum and maximum write range, and the minimum value is nonzero.

In addition, **writev** may return this error:

EINVAL *iovcnt* was less than or equal to 0, or greater than {**IOV_MAX**}.

One of the **iov_len** values in the *iov* array was negative.

The sum of the **iov_len** values in the *iov* array overflowed an **int**.

In addition, **pwrite** fails and the file pointer remains unchanged if this is true:

ESPIPE *fildes* is associated with a pipe or FIFO.

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

Upon successful completion **write** and **writev** mark for update the **st_ctime** and **st_mtime** fields of the file.

In addition, **writel**, **pwritel**, and **writevl** may set **errno** to

EFAULT *label_p* points outside the allocated address space of the process. The seek pointer remains unchanged if this error occurs.

EPERM The calling process attempted to upgrade the information label associated with the file-system object and did not have **PRIV_FILE_UPGRADE_IL** in its set of effective privileges.

The calling process attempted to downgrade the information label associated with the file-system object and did not have **PRIV_FILE_DOWNGRADE_IL** in its set of effective privileges.

SUMMARY OF TRUSTED SOLARIS CHANGES

If set-user-ID or get-group-ID permission bits of *fildes* are set, they are cleared by the write. The calling process may assert the **PRIV_FILE_SETID** privilege to suppress this action.

If the forced or allowed privilege set of *fildes* is not empty, it is cleared by the write. The calling process may assert the **PRIV_FILE_SETPRIV** privilege to suppress this action.

If the public object attributes flag, FAF_PUBLIC, of *fildes* is set, the flag is cleared by the write. The calling process may assert the **PRIV_FILE_AUDIT** privilege to suppress this action.

If the write causes the file-system free space to fall below its minimum level, the write fails. The calling process may assert the **PRIV_SYS_MINFREE** privilege to bypass this restriction.

Mandatory and discretionary access checks have already been performed when the object was opened.

When the calling process writes to a regular file, the process information label floats the information label of the file unless the **PRIV_FILE_NOFLOAT** privilege is asserted. For conduits, the information label of the calling process is associated with each byte of data in the conduit unless the **PRIV_FILE_NOFLOAT** privilege is asserted.

SEE ALSO

Intro(2TSOL), **chmod**(2TSOL), **creat**(2TSOL), **dup**(2), **fcntl**(2TSOL), **getrlimit**(2TSOL), **ioctl**(2), **lseek**(2TSOL), **open**(2TSOL), **pipe**(2), **ulimit**(2TSOL), **socket**(3N), **streamio**(7I)

Index