man pages section 2: System Calls

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 835-8003
December 2000

Contents

Preface

Overview

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

Trusted Solaris Reference Manual

In the AnswerBook2™ and online man command forms of the man pages, all man pages are available:

- Trusted Solaris man pages that are unique for the Trusted Solaris environment
- SunOS 5.8 man pages that have been changed in the Trusted Solaris environment
- SunOS 5.8 man pages that remain unchanged.

The printed manual, the *Trusted Solaris 8 Reference Manual* contains:

- Man pages that have been added to the SunOS operating system by the Trusted Solaris environment
- Man pages that originated in SunOS 5.8, but have been modified in the Trusted Solaris environment to handle security requirements.

Users of printed manuals need both manuals in order to have a full set of man pages, since the *SunOS 5.8 Reference Manual* contains the common man pages that are not modified in the Trusted Solaris environment.

Man Page Sections

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

- Section 1 describes, in alphabetical order, commands available with the operating system.
- Section 1M describes, in alphabetical order, commands that are used chiefly for system maintenance and administration purposes.
- Section 2 describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value.
- Section 3 describes functions found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2 of this volume.
- Section 4 outlines the formats of various files. The C structure declarations for the file formats are given where applicable.
- Section 5 contains miscellaneous documentation such as character set tables.
- Section 6 contains available games and demos.
- Section 7 describes various special files that refer to specific hardware peripherals, and device drivers. STREAMS software drivers, modules and the STREAMS-generic set of system calls are also described.
- Section 9 provides reference information needed to write device drivers in the kernel operating systems environment. It describes two device driver interface specifications: the Device Driver Interface (DDI) and the Driver/Kernel Interface (DKI).
- Section 9E describes the DDI/DKI, DDI-only, and DKI-only entry-point routines a developer may include in a device driver.
- Section 9F describes the kernel functions available for use by device drivers.
- Section 9S describes the data structures used by drivers to share information between the driver and the kernel.

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

NAME

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

SYNOPSIS

This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full pathname is shown. 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 a time.

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

PROTOCOL

This section occurs only in subsection 3R to indicate the protocol description file.

DESCRIPTION

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

IOCTL

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

OPTIONS

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

OPERANDS

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

OUTPUT

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

RETURN VALUES

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

ERRORS

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

USAGE

This section lists special rules, features, and commands that require in-depth explanations. The subsections listed here are used to explain built-in functionality:

- Commands
- Modifiers
- Variables
- Expressions
- Input Grammar

EXAMPLES

This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command-line entry and machine response is shown. Whenever an example is given, the prompt is shown as `example%`, or if the user must be root, `example#`. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS, and USAGE sections.

ENVIRONMENT VARIABLES

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

EXIT STATUS

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

FILES

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

ATTRIBUTES

This section lists characteristics of commands, utilities, and device drivers by defining the attribute type and its corresponding value. See `attributes`(5) for more information.

SUMMARY OF TRUSTED SOLARIS CHANGES

This section describes changes to a Solaris item by Trusted Solaris software. It is present in man pages that have been modified from Solaris software.

SEE ALSO

This section lists references to other man pages, in-house documentation and outside publications. The references are divided into two sections, so that users of printed manuals can easily locate a man page in its appropriate printed manual.

DIAGNOSTICS

This section lists diagnostic messages with a brief explanation of the condition causing the error.

WARNINGS

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

NOTES

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

BUGS

This section describes known bugs and, wherever possible, suggests workarounds.

CHAPTER

System Calls

24

NAME

Intro – Introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

This section describes all of the system calls in the Trusted Solaris environment.

Trusted Solaris system calls are one of the following:

- Calls that are unique to and originate in the Trusted Solaris environment, such as secconf(2). 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.
- SunOS 5.8 system calls that have been modified to work within Trusted Solaris security policy, such as link(2). 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 environment. Modified man pages also have added descriptions for new features and arguments.
- SunOS 5.8 system calls that remain unchanged from the Solaris 8 release, such as exit(2).

The printed *Trusted Solaris 8 Reference Manual* includes only those system calls that have been modified or originate in the Trusted Solaris environment. Printed versions of unchanged SunOS 5.8 man pages are found in the *SunOS 5.8 Reference Manual*. For more discussion of Trusted Solaris Manual Page Display, see Trusted Solaris Manual Page Display in Intro(1).

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 section, and see also the *Trusted Solaris Developer's Guide* for more complete descriptions of the topics mentioned here.

ERRORS

Most of these calls return one or more error conditions. An error condition is indicated by an otherwise impossible return value. This is almost always –1 or the null pointer; the individual descriptions specify the details. An error number

is also made available in the external variable `errno`, which is not cleared on successful calls, so it should be tested only after an error has been indicated.

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

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

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

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 section.
2 ENOENT	No such file or directory A file name is specified and the file should exist but doesn't, or one of the directories in a path name does not exist.
3 ESRCH	No such process, LWP, or thread No process can be found in the system that corresponds to the specified PID, `LWPID_t`, or `thread_t`.
4 EINTR	Interrupted system call An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system service function. If execution is resumed after processing the signal, it will appear as if the interrupted function call returned this error condition. In a multithreaded application, `EINTR` may be returned whenever another thread or LWP calls `fork`(2).

5 EIO	I/O error Some physical I/O error has occurred. This error may in some cases occur on a call following the one to which it actually applies.
6 ENXIO	No such device or address I/O on a special file refers to a subdevice which does not exist, or exists beyond the limit of the device. It may also occur when, for example, a tape drive is not on-line or no disk pack is loaded on a drive.
7 E2BIG	Arg list too long An argument list longer than `ARG_MAX` bytes is presented to a member of the `exec` family of functions (see `exec`(2)). The argument list limit is the sum of the size of the argument list plus the size of the environment's exported shell variables.
8 ENOEXEC	Exec format error A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid format (see `a.out`(4)).
9 EBADF	Bad file number Either a file descriptor refers to no open file, or a `read`(2) (respectively, `write`(2)) request is made to a file that is open only for writing (respectively, reading).
10 ECHILD	No child processes A `wait`(2) function was executed by a process that had no existing or unwaited-for child processes.
11 EAGAIN	No more processes, or no more LWPs For example, the `fork`(2) function failed because the system's process table is full or the user is not allowed to create any more processes, or a call failed because of insufficient memory or swap space.
12 ENOMEM	Not enough space

During execution of `brk()` or `sbrk()` (see `brk`(2)), or one of the `exec` family of functions, a program asks for more space than the system is able to supply. This is not a temporary condition; the maximum size is a system parameter. On some architectures, the error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers, or if there is not enough swap space during the `fork`(2) function. If this error occurs on a resource associated with Remote File Sharing (RFS), it indicates a memory depletion which may be temporary, dependent on system activity at the time the call was invoked.

13 EACCES — Permission denied

An attempt was made to access a file in a way forbidden by the 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 depends on the type of access being denied). See *Discretionary Access Control*, *File Access*, *Mandatory Access Control*, *Privilege*, and *Security Policy* in the DEFINITIONS section.

14 EFAULT — Bad address

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

15 ENOTBLK — Block device required

A non-block device or file was mentioned where a block device was required (for example, in a call to the `mount`(2) function).

16 EBUSY — Device busy

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

17 EEXIST — File exists

An existing file was mentioned in an inappropriate context (for example, call to the `link`(2) function).

18 EXDEV — Cross-device link

A hard link to a file on another device was attempted.

19 ENODEV — No such device

An attempt was made to apply an inappropriate operation to a device (for example, read a write-only device).

20 ENOTDIR — Not a directory

A non-directory was specified where a directory is required (for example, in a path prefix or as an argument to the `chdir`(2) function).

21 EISDIR — Is a directory

An attempt was made to write on a directory.

22 EINVAL — Invalid argument

An invalid argument was specified (for example, unmounting a non-mounted device), mentioning

	an undefined signal in a call to the `signal`(3C) or `kill`(2) function.
23 ENFILE	File table overflow
	The system file table is full (that is, `SYS_OPEN` files are open, and temporarily no more files can be opened).
24 EMFILE	Too many open files
	No process may have more than `OPEN_MAX` file descriptors open at a time.
25 ENOTTY	Inappropriate ioctl for device
	A call was made to the `ioctl`(2) function specifying a file that is not a special character device.
26 ETXTBSY	Text file busy (obselete)
	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`; the file size exceeds the maximum supported by the file system; or the file size exceeds the offset maximum of the file descriptor. See the `File Descriptor` subsection of the `DEFINITIONS` section below.
28 ENOSPC	No space left on device
	While writing an ordinary file or creating a directory entry, there is no free space left on the device. In the `fcntl`(2) function, the setting or removing of record locks on a file cannot be accomplished because there are no more record entries left on the system.
29 ESPIPE	Illegal seek
	A call to the `lseek`(2) function was issued to a pipe.

30 EROFS	Read-only file system An attempt to modify a file or directory was made on a device mounted read-only.
31 EMLINK	Too many links An attempt to make more than the maximum number of links, `LINK_MAX`, to a file.
32 EPIPE	Broken pipe A write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.
33 EDOM	Math arguement 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 `msgrcv`(2)).
36 EIDRM	Identifier removed This error is returned to processes that resume execution due to the removal of an identifier from the file system's name space (see `msgctl`(2), `semctl`(2), and `shmctl`(2)).
37 ECHRNG	Channel number out of range
38 EL2NSYNC	Level 2 not synchronized
39 EL3HLT	Level 3 halted
40 EL3RST	Level 3 reset
41 ELNRNG	Link number out of range
42 EUNATCH	Protocol driver not attached
43 ENOCSI	No CSI structure available

44 EL2HLT	Level 2 halted
45 EDEADLK	Deadlock condition A deadlock situation was detected and avoided. This error pertains to file and record locking, and also applies to mutexes, semaphores, condition variables, and r/w locks.
46 ENOLCK	No record locks available There are no more locks available. The system lock table is full (see `fcntl`(2)).
47 ECANCELED	Operation canceled The associated asynchronous operation was canceled before completion.
48 ENOTSUP	Not supported This version of the system does not support this feature. Future versions of the system may provide support.
49 EDQUOT	Disc quota exceeded A `write`(2) to an ordinary file, the creation of a directory or symbolic link, or the creation of a directory entry failed because the user's quota of disk blocks was exhausted, or the allocation of an inode for a newly created file failed because the user's quota of inodes was exhausted.
58-59	Reserved
60 ENOSTR	Device not a stream A `putmsg`(2) or `getmsg`(2) call was attempted on a file descriptor that is not a STREAMS device.
61 ENODATA	No data available
62 ETIME	Timer expired The timer set for a STREAMS `ioctl`(2) call has expired. The cause of this error is device-specific and could indicate either a hardware or software failure, or perhaps a timeout value that is too short for the specific operation. The status of the `ioctl( )` operation

	is indeterminate. This is also returned in the case of `_lwp_cond_timedwait`(2) or `_lwp_cond_wait`(2).
63 ENOSR	Out of stream resources
	During a STREAMS `open`(2) call, either no STREAMS queues or no STREAMS head data structures were available. This is a temporary condition; one may recover from it if other processes release resources.
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 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.

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

`read()`:	control information or passed file descriptor.
`getmsg()`:	passed file descriptor.
`ioctl()`:	control or data information.

78 ENAMETOOLONG — File name too long

The length of the path argument exceeds `PATH_MAX`, or the length of a path component exceeds `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect; see `limits`(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 share library
	Trying to `exec` an `a.out` that requires a static shared library and the static shared library does not exist or the user does not have permission to use it.
84 ELIBBAD	Accessing a corrupted shared library
	Trying to `exec` an `a.out` that requires a static shared library (to be linked in) and `exec` could not load the static shared library. The static shared library is probably corrupted.
85 ELIBSCN	`.lib` section in `a.out` corrupted
	Trying to `exec` an `a.out` that requires a static shared library (to be linked in) and there was erroneous data in the `.lib` section of the `a.out`. The `.lib` section tells `exec` what static shared libraries are needed. The `a.out` is probably corrupted.
86 ELIBMAX	Attempting to link in more shared libraries than system limit
	Trying to `exec` an `a.out` that requires more static shared libraries than is allowed on the current configuration of the system. See *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 EMGSIZE	Message too long
	A message sent on a transport provider was larger than the internal message buffer or some other network limit.
98 EPROTOTYPE	Protocol wrong type for socket
	A protocol was specified that does not support the semantics of the socket type requested.
99 ENOPROTOOPT	Protocol not available
	A bad option or level was specified when getting or setting options for a protocol.
120 EPROTONOSUPPORT	Protocol not supported
	The protocol has not been configured into the system or no implementation for it exists.
121 ESOCKTNOSUPPORT	Socket type not supported
	The support for the socket type has not been configured into the system or no implementation for it exists.
122 EOPNOTSUPP	Operation not supported on transport endpoint
	For example, trying to accept a connection on a datagram transport endpoint.
123 EPFNOSUPPORT	Protocol family not supported

	The protocol family has not been configured into the system or no implementation for it exists. Used for the Internet protocols.
124 EAFNOSUPPORT	Address family not supported by protocol family
	An address incompatible with the requested protocol was used.
125 EADDRINUSE	Address already in use
	User attempted to use an address already in use, and the protocol does not allow this.
126 EADDRNOTAVAIL	Cannot assign requested address
	Results from an attempt to create a transport endpoint with an address not on the current machine.
127 ENETDOWN	Network is down
	Operation encountered a dead network.
128 ENETUNREACH	Network is unreachable
	Operation was attempted to an unreachable network.
129 ENETRESET	Network dropped connection because of reset
	The host you were connected to crashed and rebooted.
130 ECONNABORTED	Software caused connection abort
	A connection abort was caused internal to your host machine.
131 ECONNRESET	Connection reset by peer
	A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote host due to a timeout or a reboot.
132 ENOBUFS	No buffer space available
	An operation on a transport endpoint or pipe was not performed because the system lacked sufficient buffer space or because a queue was full.
133 EISCONN	Transport endpoint is already connected

	A connect request was made on an already connected transport endpoint; or, a `sendto`(3SOCKET) or `sendmsg`(3SOCKET) request on a connected transport endpoint specified a destination when already connected.
134 ENOTCONN	Transport endpoint is not connected
	A request to send or receive data was disallowed because the transport endpoint is not connected and (when sending a datagram) no address was supplied.
143 ESHUTDOWN	Cannot send after transport endpoint shutdown
	A request to send data was disallowed because the transport endpoint has already been shut down.
144 ETOOMANYREFS	Too many references: cannot splice
145 ETIMEDOUT	Connection timed out
	A `connect`(3SOCKET) or `send`(3SOCKET) request failed because the connected party did not properly respond after a period of time; or a `write`(2) or `fsync`(3C) request failed because a file is on an NFS file system mounted with the *soft* option.
146 ECONNREFUSED	Connection refused
	No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the remote host.
147 EHOSTDOWN	Host is down
	A transport provider operation failed because the destination host was down.
148 EHOSTUNREACH	No route to host
	A transport provider operation was attempted to an unreachable host.
149 EALREADY	Operation already in progress
	An operation was attempted on a non-blocking object that already had an operation in progress.

150 EINPROGRESS	Operation now in progress An operation that takes a long time to complete (such as a `connect()`) was attempted on a non-blocking object.
151 ESTALE	Stale NFS file handle

DEFINITIONS

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 environment.

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 Each process has a clearance associated with it. A *clearance* consists of a *classification* and a set of *compartments.* It is similar to a *sensitivity label*.A process' clearance is an upper bound on the labels to which the process has access. A process can neither set its sensitivity label to a label that dominates its clearance, nor access an object (file or other process) whose sensitivity label dominates the process clearance.

Compartment A word associated with one or more compartment bits that may be defined in the `label_encodings`(4) 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 access 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 catalogs the list of files, including directories (sub-directories), that are directly beneath it in the hierarchy. Entries in a directory file are called links. A link associates a file identifier with a filename. By convention, a directory contains at least two links, `.` (dot) and `..` (dot-dot). The link called dot refers to the directory itself while dot-dot refers to its parent directory. The root directory, which is the top-most node of the hierarchy, has itself as its parent directory. The pathname of the root directory is `/` and the parent directory of the root directory is `/`.

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 environment 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 environment, 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 security policy for device files can differ from the policy for regular files based on how the policy is defined in the `device_policy`(4) file, which can be changed by the security administrator. 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 `PRIV_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 `PRIV_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 `PRIV_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 `PRIV_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 label 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 `PRIV_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 `PRIV_FILE_DAC_WRITE` .

File Access Permissions

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

- If the effective UID of the process matches the UID of the file, access is granted if allowed by the file's "owner" permission set.
- If the effective UID of the process matches an ACL user entry, access is granted if allowed by the ACL entry and the ACL mask.
- If the effective GID of the process (or one of its supplemental groups) matches the GID of the file or the group ID of any ACL group entry, a permission set is computed as the inclusive `OR` of all matching group permission sets, specified as follows:
  - If the effective GID of the process (or one of its supplemental groups) matches the GID of the file and there is no owning group ACL entry for the file, the "group" permission set is considered a matching group permission set.
  - If the effective GID of the process (or one of its supplemental groups) matches the GID of the file and there is an owning group ACL entry for the file, the permissions set of that ACL entry is considered a matching group permission set.
  - If the effective GID of the process (or one of its supplemental groups) matches an ACL group entry, the permission set of that ACL entry is considered a matching group permission set.

  Access is granted if allowed by the computed permission set and the ACL mask, if any.
- If none of the preceding cases applies, access is granted if allowed by the file's "other" permission set and the ACL mask, if any.

A process is granted access if it satisfies the appropriate test above or if it asserts the DAC override privilege corresponding to the desired operation. Otherwise, access is denied.

File Descriptor

A file descriptor is a small integer used to perform I / O on a file. The value of a file descriptor is from `0` to (`NOFILES`–`1`). A process may have no more than `NOFILES` file descriptors open simultaneously. A file descriptor is returned by calls such as `open`(2) or `pipe`(2). The file descriptor is used as an argument by calls such as `read`(2), `write`(2), `ioctl`(2), and `close`(2).

Each file descriptor has a corresponding offset maximum. For regular files that were opened without setting the `O_LARGEFILE` flag, the offset maximum

is 2 Gbyte – 1 byte (2^{31} –1 bytes). For regular files that were opened with the `O_LARGEFILE` flag set, the offset maximum is 2^{63} –1 bytes.

File Name

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

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

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

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

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

An *information label* ideally represents the present actual classification and compartments of, and any required markings or handling that apply to, the data with which it is associated. An information label consists of a hierarchical classification and a set of non-hierarchical compartments. This

classification-and-compartments pair is known as the information level of the information label. In addition to the information level, an information label has a set of non-hierarchical markings. Note that information labels are no longer supported in the Trusted Solaris environment.

Inheritable Privileges The privileges that a process can pass to a program across an `execve`(2) 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 labels 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`(1M).] 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 The label view process attribute flags control the translation and display of the internal `admin low` and `admin high` labels. A value of `External` specifies that the `admin low` and `admin high` labels are mapped to the lowest and highest labels defined in the `label_encodings`(4) file. A value of `Internal` specifies that the `admin low` and `admin high` labels are translated to the respective strings specified in the `label_encodings` file. If no such names are specified, the strings `ADMIN_LOW` and `ADMIN_HIGH` are used. If no value is set, the default label view specified in the `label_encodings` file is used.

Label Translation Flags These fifteen-bit flags support the GFI FLAGS= option in the `label_encodings`(4) 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.

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

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

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

Message Queue Identifier A message queue identifier (`msqid`) is a unique positive integer created by a `msgget`(2) call. Each `msqid` has a message queue and a data structure associated with it. The data structure is referred to as `msqid_ds` and contains the following members:

```
struct     ipc_perm msg_perm;
struct     msg *msg_first;
struct     msg *msg_last;
ulong_t    msg_cbytes;
ulong_t    msg_qnum;
ulong_t    msg_qbytes;
pid_t      msg_lspid;
pid_t      msg_lrpid;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Message Operation Permissions

In the `msgctl`(2), `msgget`(2), `msgrcv`(2), and `msgsnd`(2) function descriptions, the permission required for an operation is given as {*token*}, where *token* is the type of permission needed, interpreted as follows:

```
00400   READ by user
00200   WRITE by user
00040   READ by group
00020   WRITE by group
```

```
00004   READ by others
00002   WRITE by others
```

Read and write permissions 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:

- The effective user ID of the process matches `msg_perm.cuid` or `msg_perm.uid` in the data structure associated with `msqid` and the appropriate bit of the "user" portion (0600) of `msg_perm.mode` is set.
- Any group ID in the process credentials from the set (`cr_gid`, `cr_groups`) matches `msg_perm.cgid` or `msg_perm.gid` and the appropriate bit of the "group" portion (060) of `msg_perm.mode` is set.
- The appropriate bit of the "other" portion (006) of `msg_perm.mode` is set.
- 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 environment, 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, or by using the adorned name. When a process refers to an MLD without the adorned name, the Trusted Solaris process 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`(1M). 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 environment 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.

Offset Maximum

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

Orphaned Process Group

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

Path Name

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

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

A slash by itself names the root directory.

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

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 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`(1M) when executing a command in privilege debugging mode, and then is inherited by the process. It works only if the `_PRIVS_DEBUG` kernel switch is also enabled (see `secconf`(2)).

Process Attribute Flags

Trusted Solaris flags that indicate security-related values that are copied from one process to another on `fork`(2) and cloned without changes on `exec`(2). 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`(1), `getpattr`(2) , and `setpattr`(2). 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 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`(2) 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 compatibility with the base system's super-user capability, if the effective UID is set by `setuid`(2) 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`(2) and `setppriv`(2) 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`(2) 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 Attributes

Security attributes received by processes from the Solaris operating environment are: the process ID (PID), the real, effective, and saved user ID, the real, effective, or saved group 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.

Processor Set ID

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

Read Queue

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

Real User ID and Real Group ID

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

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

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

Root Directory and Current Working Directory

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

Saved Resource Limits

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

Saved User ID and Saved Group ID

The saved user ID and saved group ID are the values of the effective user ID and effective group ID 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 Solaris operating environment and the Trusted Solaris operating environment, 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, 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`(2) call. Each `semid` has a set of semaphores and a data structure associated with it. The data structure is referred to as `semid_ds` and contains the following members:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Semaphore Operation Permissions

In the `semop`(2) and `semctl`(2) function descriptions, the permission required for an operation is given as {*token*}, where *token* is the type of permission needed interpreted as follows:

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

Read and alter permissions 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.

- The effective user ID of the process matches `sem_perm.cuid` or `sem_perm.uid` in the data structure associated with `semid` and the appropriate bit of the "user" portion (0600) of `sem_perm.mode` is set.
- The effective group ID of the process matches `sem_perm.cgid` or `sem_perm.gid` and the appropriate bit of the "group" portion (060) of `sem_perm.mode` is set.
- The appropriate bit of the "other" portion (06) of `sem_perm.mode` is set.
- 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 *sensitivity label* defines the level of protection afforded to a labeled object or the level of access granted a labeled subject. Sensitivity labels are used in all mandatory access control (MAC) decisions by the Trusted Solaris environment. A sensitivity label consists of a hierarchical classification and a set of non-hierarchical compartments. This classification-and-compartments pair is known as the level of the sensitivity label.

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 `semget`(2) call. Each `shmid` has a segment of memory (referred to as a shared memory segment) and a data structure associated with it. (Note that these shared memory segments must be explicitly removed by the user after the last reference to them is removed.) The data structure is referred to as `shmid_ds` and contains the following members:

```
struct ipc_perm   shm_perm;     /* operation permission struct */
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_t          shm_nattch;   /* number of current attaches */
ushort_t          shm_cnattch;  /* used only for shminfo */
time_t            shm_atime;    /* last attach time */
time_t            shm_dtime;    /* last detach time */
time_t            shm_ctime;    /* last change time */
                                /* Times measured in secs since */
                                /* 00:00:00 GMT, Jan. 1, 1970 */
```

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

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

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

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

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

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

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

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

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

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

Shared Memory Operation Permissions

In the shmctl(2), shmat( ), and shmdt( ) (see shmop(2)) function descriptions, the permission required for an operation is given as {*token*}, where *token* is the type of permission needed interpreted as follows:

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

Read and write permissions for a shmid are granted to a process if 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:

- The effective user ID of the process is super-user.
- The effective user ID of the process matches shm_perm.cuid or shm_perm.uid in the data structure associated with shmid and the appropriate bit of the "user" portion (0600) of shm_perm.mode is set.
- The effective group ID of the process matches shm_perm.cgid or shm_perm.gid and the appropriate bit of the "group" portion (060) of shm_perm.mode is set.
- The appropriate bit of the "other" portion (06) of shm_perm.mode is set.
- 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 SL 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*>/. All subsequent references to the file in the /tmpdirectory 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 MLD 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 principal functions of the stream head are processing STREAMS-related system calls and passing data and information between a user process and the stream.

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 dominates 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 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`(4) file, plus the two administrative labels that are used in every Trusted Solaris environment, `ADMIN_LOW` and `ADMIN_HIGH`.

Superuser A process is recognized as a superuser process and is granted special privileges, such as immunity from file permissions, if its effective user ID is 0. In the Trusted Solaris environment, superuser is replaced by administrative roles that share responsibility for the environment.

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 `/usr/openwin/man/man3`) for exceptions.

Name	Description
`access`(2)	Determine accessibility of a file
`acct`(2)	Enable or disable process accounting
`acl`(2)	Get or set a file's Access Control List (ACL)
`adjtime`(2)	Correct the time to allow synchronization of the system clock
`audit`(2)	Write a record to the audit log
`auditon`(2)	Manipulate auditing
`auditsvc`(2)	Write audit log to specified file descriptor
`chdir`(2)	Change working directory
`chmod`(2)	Change access permission mode of file
`chown`(2)	Change owner and group of a file
`chroot`(2)	Change root directory
`chstate`(2)	Change the view of a host state between labeled and unlabeled
`creat`(2)	Create a new file or rewrite an existing one
`devpolicy`(2)	Get/set device driver policy table
`exec`(2)	execute a file
`execl`(2)	See `exec`(2)
`execle`(2)	See `exec`(2)
`execlp`(2)	See `exec`(2)
`execv`(2)	See `exec`(2)
`execve`(2)	See `exec`(2)

execvp(2)	See exec(2)
facl(2)	See acl(2)
fchdir(2)	See chdir(2)
fchmod(2)	See chmod(2)
fchown(2)	See chown(2)
fchroot(2)	See chroot(2)
fcntl(2)	File control
fgetcmwfsrange(2)	See getcmwfsrange(2)
fgetcmwlabel(2)	See getcmwlabel(2)
fgetfattrflag(2)	See getfattrflag(2)
fgetfpriv(2)	See getfpriv(2)
fgetfsattr(2)	See getfsattr(2)
fgetmldadorn(2)	See getmldadorn(2)
fgetsldname(2)	See getsldname(2)
fork(2)	Create a new process
fork1(2)	See fork(2)
fpathconf(2)	Get configurable pathname variables
fsetcmwlabel(2)	See setcmwlabel(2)
fsetfattrflag(2)	See getfattrflag(2)
fsetfpriv(2)	See getfpriv(2)
fstat(2)	See stat(2)
fstatvfs(2)	See statvfs(2)
getaudit(2)	Get and set process audit information
getaudit_addr(2)	See getaudit(2)
getauid(2)	Get and set user audit identity
getclearance(2)	Get process clearance
getcmwfsrange(2)	Get file system sensitivity label range
getcmwlabel(2)	Get file CMW label
getcmwplabel(2)	Get process CMW label

getdents(2)	Read directory entries and put in a file system independent format
getfattrflag(2)	Set / get the security attribute flags of a file
getfpriv(2)	Return or set a privilege set associated with a file
getfsattr(2)	Get filesystem security attributes
getgroups(2)	Get or set supplementary group access list IDs
getmldadorn(2)	Get file system multilevel directory adornment
getmsgqcmwlabel(2)	Get the CMW labels associated with System V IPC structures
getpattr(2)	Get / set process attribute flags
getpgid(2)	See getpid(2)
getpgrp(2)	See getpid(2)
getpid(2)	Get process, process group, and parent process IDs
getppid(2)	See getpid(2)
getppriv(2)	Return or assign a privilege set associated with the invoking process
getrlimit(2)	Control maximum system resource consumption
getsemcmwlabel(2)	See getmsgqcmwlabel(2)
getshmcmwlabel(2)	See getmsgqcmwlabel(2)
getsid(2)	Get process group ID of session leader
getsldname(2)	Get file system single-level directory name
kill(2)	Send a signal to a process or a group of processes
lchown(2)	See chown(2)
lgetcmwlabel(2)	See getcmwlabel(2)
link(2)	Link to a file
llseek(2)	Move extended read / write file pointer
lseek(2)	Move read / write file pointer
lsetcmwlabel(2)	See setcmwlabel(2)
lstat(2)	See stat(2)

mkdir(2)	Make a directory
mknod(2)	Make a directory, or a special or ordinary file
mldgetfattrflag(2)	See getfattrflag(2)
mldsetfattrflag(2)	See getfattrflag(2)
mount(2)	Mount a file system
msgctl(2)	Message control operations
msgget(2)	Get message queue
msggetl(2)	See msgget(2)
msgrcv(2)	Message receive operation
msgsnd(2)	Message send operation
nice(2)	Change priority of a process
open(2)	Open a file
p_online(2)	Return or change processor operational status
pathconf(2)	See fpathconf(2)
pread(2)	See read(2)
preadl(2)	See read(2)
priocntl(2)	Process scheduler control
priocntlset(2)	Generalized process scheduler control
processor_bind(2)	Bind LWPs to a processor
pwrite(2)	See write(2)
pwritel(2)	See write(2)
read(2)	Read from a file
readl(2)	See read(2)
readlink(2)	Read the contents of a symbolic link
readv(2)	See read(2)
readvl(2)	See read(2)
rename(2)	Change the name of a file
rmdir(2)	Remove a directory
secconf(2)	Get security configuration information

semctl(2)	Semaphore control operations
semget(2)	Get set of semaphores
semgetl(2)	See semget(2)
semop(2)	Semaphore operations
setaudit(2)	See getaudit(2)
setaudit_addr(2)	See getaudit(2)
setauid(2)	See getauid(2)
setclearance(2)	Set process clearance
setcmwlabel(2)	Set CMW label of a file
setcmwplabel(2)	Set process CMW label
setegid(2)	See setuid(2)
seteuid(2)	See setuid(2)
setfattrflag(2)	See getfattrflag(2)
setfpriv(2)	See getfpriv(2)
setgid(2)	See setuid(2)
setgroups(2)	See getgroups(2)
setpattr(2)	See getpattr(2)
setppriv(2)	See getppriv(2)
setregid(2)	Set real and effective group IDs
setreuid(2)	Set real and effective user IDs
setrlimit(2)	See getrlimit(2)
setuid(2)	Set user and group IDs
shmat(2)	See shmop(2)
shmctl(2)	Shared memory control operations
shmdt(2)	See shmop(2)
shmget(2)	Get shared memory segment identifier
shmgetl(2)	See shmget(2)
shmop(2)	Shared memory operations
sigsend(2)	Send a signal to a process or a group of processes

sigsendset(2)	See sigsend(2)
stat(2)	Get file status
statvfs(2)	Get file system information
stime(2)	Set system time and date
swapctl(2)	Manage swap space
symlink(2)	Make a symbolic link to a file
sysinfo(2)	Get and set system information strings
tokmapper(2)	Manipulate kernel token mapping caches
uadmin(2)	Administrative control
ulimit(2)	Get and set process limits
umount(2)	Unmount a file system
umount2(2)	See umount(2)
unlink(2)	Remove directory entry
utimes(2)	Set file access and modification times
vfork(2)	Spawn new process in a virtual memory efficient way
write(2)	Write on a file
writel(2)	See write(2)
writev(2)	See write(2)
writevl(2)	See write(2)

NAME

access – Determine accessibility of a file

SYNOPSIS

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

DESCRIPTION

The `access()` function checks the file named by the pathname pointed to by the *path* argument for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID. This allows a setuid process to verify that the user running it would have had permission to access this file.

The value of *amode* is either the bitwise inclusive OR of the access permissions to be checked (`R_OK`, `W_OK`, `X_OK`) or the existence test, `F_OK`.

These constants are defined in `<unistd.h>` as follows:

`R_OK` Test for read permission.

`W_OK` Test for write permission.

`X_OK` Test for execute or search permission.

`F_OK` Check existence of file

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

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

RETURN VALUES

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

ERRORS

The `access()` function will fail if:

`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 resolving *path*.
ENAMETOOLONG	The length of the *path* argument exceeds PATH_MAX, or a pathname component is longer than NAME_MAX while _POSIX_NO_TRUNC is in effect.
ENOENT	A component of *path* does not name an existing file or *path* is an empty string.
ENOLINK	*path* points to a remote machine and the link to that machine is no longer active.
ENOTDIR	A component of the path prefix is not a directory.
EROFS	Write access is requested for a file on a read-only file system.

The access() function may fail if:

EINVAL	The value of the *amode* argument is invalid.
ENAMETOOLONG	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.
ETXTBSY	Write access is requested for a pure procedure (shared text) file that is being executed.

USAGE

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

ATTRIBUTES

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

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

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

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
Trusted Solaris 8 Reference Manual

intro(2), chmod(2), stat(2)

SunOS 5.8 Reference Manual

`attributes`(5)

NAME

acct – Enable or disable process accounting

SYNOPSIS

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

DESCRIPTION

The `acct()` function enables or disables the system process accounting routine. If the routine is enabled, an accounting record will be written in an accounting file for each process that terminates. The termination of a process can be caused by either an `exit`(2) call or a `signal`(3C). The effective privilege set of the process calling `acct()` must include `PRIV_SYS_CONFIG`.

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

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

RETURN VALUES

`acct()` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `acct()` function will fail if:

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

SUMMARY OF TRUSTED SOLARIS CHANGES

The effective privilege set of the process calling `acct()` must include `PRIV_SYS_CONFIG`.

SEE ALSO

SunOS 5.8 Reference Manual

`exit`(2), `signal`(3C), `acct`(3HEAD)

NAME

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

SYNOPSIS

```
#include <sys/acl.h>
int acl(char * pathp, int cmd, int nentries, aclent_t * aclbufp);

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

DESCRIPTION

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

The following values for *cmd* are supported:

`SETACL` *nentries* ACL entries, specified in buffer *aclbufp* , are stored in the file's ACL . This command can 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.

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

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

RETURN VALUES

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

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

ERRORS

The `acl()` function will fail if:

`EACCESS` The caller does not have 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` .

The *cmd* argument is `SETACL` and *nentries* is less than three.

The *cmd* argument is `SETACL` and the ACL specified in *aclbufp* is not valid.

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

`EINVAL` The *cmd* argument is not `GETACL` , `SETACL` , or `GETACLCNT` ; the *cmd* argument is `SETACL` and *nentries* is less than 3;

or the *cmd* argument is SETACL and the ACL specified in *aclbufp* is not valid.

EIO — A disk I/O error has occurred while storing or retrieving the ACL .

ENOENT — A component of the path does not exist.

ENOSPC — The *cmd* argument is GETACL and *nentries* is less than the number of entries in the file's ACL , or the *cmd* argument is SETACL and there is insufficient space in the file system to store the ACL .

ENOTDIR — A component of the path specified by *pathp* is not a directory, or the *cmd* argument is SETACL and an attempt is made to set a default ACL on a file type other than a directory.

ENOSYS — The *cmd* argument is SETACL and the file specified by *pathp* resides on a file system that does not support ACLs, or the acl( ) function is not supported by this implementation.

EPERM — The *cmd* argument is SETACL and the effective user ID of the caller does not match the owner of the file. To override this restriction, the calling process may assert the PRIV_FILE_SETDAC privilege.

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access or ownership checks.

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

SEE ALSO

SunOS 5.8 Reference Manual

getfacl(1), setfacl(1), aclcheck(3SEC), aclsort(3SEC), attributes(5)

NAME

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

SYNOPSIS

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

DESCRIPTION

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

The adjustment is effected by speeding up (if that amount of time is positive) or slowing down (if that amount of time is negative) the system's clock by some small percentage, generally a fraction of one percent. The time is always a monotonically increasing function. A time correction from an earlier call to `adjtime( )` may not be finished when `adjtime( )` is called again.

If *delta* is 0, then *olddelta* returns the status of the effects of the previous `adjtime( )` call with no effect on the time correction as a result of this call. If *olddelta* is not a null pointer, then the structure it points to will contain, upon successful return, the number of seconds and / or microseconds still to be corrected from the earlier call. If *olddelta* is a null pointer, the corresponding information will not be returned.

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

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

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

ERRORS

The `adjtime( )` function will fail if:

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

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

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

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

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

SUMMARY OF TRUSTED SOLARIS CHANGES

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

SEE ALSO

SunOS 5.8 Reference Manual

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

NAME

audit – Write a record to the audit log

SYNOPSIS

cc [*flags*…] *file* … –lbsm –lsocket –lnsl –lintl [*library*…]

```
#include <sys/param.h>
#include <bsm/audit.h>
int audit(caddr_t record, int length);
```

DESCRIPTION

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

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

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

RETURN VALUES

`audit()` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `audit()` function will fail if:

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

`EINVAL` The record header token ID is invalid or the length is either less than the header token size or greater than `MAXAUDITDATA`.

`EPERM` The process's effective privilege set does not contain the proper privilege for this operation.

SUMMARY OF TRUSTED SOLARIS CHANGES

This functionality is active only if auditing is enabled. By default, auditing is enabled in the Trusted Solaris environment. See *Trusted Solaris Audit Administration* for more information.

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.

`auditwrite`(3TSOL) is the preferred interface for creating audit records in the Trusted Solaris environment.

ATTRIBUTES

Available only on Trusted Solaris systems with auditing enabled.

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsr

SEE ALSO

Trusted Solaris 8 Reference Manual

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

SunOS 5.8 Reference Manual

`attributes`(5)

NAME

auditon – Manipulate auditing

SYNOPSIS

cc [*flags*…] *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( )` function performs various audit subsystem control operations. The *cmd* argument designates the particular audit control command. The *data* argument is a pointer to command-specific data. The *length* argument is the length in bytes of the command-specific data.

The following commands are supported:

`A_GETCOND` Return the system audit on / off / disabled condition in the integer long pointed to by *data*. The following values may be returned:

- `AUC_AUDITING` Auditing has been turned on.
- `AUC_NOAUDIT` Auditing has been turned off.
- `AUC_DISABLED` Auditing package installed, not turned on.

`A_SETCOND` Set the system's audit on / off condition to the value in the integer long to which *data* points. The following audit states may be set:

- `AUC_AUDITING` Turns on audit record generation.
- `AUC_NOAUDIT` Turns off audit record generation.

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

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

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

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

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

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

`A_SETUMASK` Set the preselection mask for all processes with the specified audit ID. The *data* argument points to the `auditinfo` structure containing the audit ID and the preselection mask. The other fields of the structure are ignored and should be set to `NULL`.

`A_SETSMASK` Set the preselection mask for all processes with the specified audit session ID. The *data* argument points to the `auditinfo` structure containing the audit session ID and the preselection mask. The other fields of the structure are ignored and should be set to `NULL`.

`A_GETQCTRL` Return the kernel audit queue control parameters. These control the high and low water marks of the number of audit records allowed in the audit queue. The high water mark is the maximum allowed number of undelivered audit records. The low water mark determines when threads blocked on the queue are wakened. Another parameter controls the size of the data buffer used by `auditsvc`(2) to write data to the audit trail. There is also a parameter that specifies a maximum delay before data is attempted to be written to the audit trail. The audit queue parameters are returned in the `au_qctrl` structure pointed to by *data*.

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

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

A_GETCWD	Return the current working directory as kept by the audit subsystem. This is a path anchored on the real root, rather than on the active root. The *data* argument points to a buffer into which the path is copied. The *length* argument is the length of the buffer.
A_GETCAR	Return the current active root as kept by the audit subsystem. This path may be used to anchor an absolute path for a path token generated by an application. The *data* argument points to a buffer into which the path is copied. The *length* argument is the length of the buffer.
A_GETSTAT	Return the system audit statistics in the audit_stat structure pointed to by *data*.
A_SETSTAT	Reset system audit statistics values. The kernel statistics value is reset if the corresponding field in the statistics structure pointed to by the *data* argument is CLEAR_VAL. Otherwise, the value is not changed.
A_SETFSIZE	Set the maximum size of an audit trail file. When the audit file reaches the designated size, it is closed and a new file started. If the maximum size is unset, the audit trail file generated by auditsvc( ) will grow to the size of the file system. The *data* argument points to the au_fstat_t structure containing the maximum audit file size in bytes. The size can not be set less than 0x80000 bytes.
A_GETFSIZE	Return the maximum audit file size and current file size in the au_fstat_t structure pointed to by the *data* argument.
A_GETPOLICY	Return the audit policy flags in the integer long pointed to by *data*.
A_SETPOLICY	Set the audit policy flags to the values in the integer long pointed to by *data*.

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.
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_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

auditon( ) returns:

0	On success.
–1	On failure, and sets errno to indicate the error.

ERRORS

The auditon( ) function will fail if:

E2BIG	The *length* field for the command was too small to hold the returned value.
EFAULT	The copy of data to/from the kernel failed.
EINVAL	One of the system call arguments was illegal..
EPERM	The process did not have the appropriate privilege in its effective set.

USAGE

The auditon( ) function may be invoked only by processes with super-user privileges.

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.

Information labels (ILs) are not supported in Trusted Solaris 7 and later releases. Trusted Solaris software interprets any ILs on communications and files from systems running earlier releases as ADMIN_LOW.

Objects still have CMW labels, and CMW labels still include the IL component: IL[SL]; however, the IL component is fixed at ADMIN_LOW.

As a result, Trusted Solaris 7 has the following characteristics:

- ILs do not display in window labels; SLs (Sensitivity Labels) display alone within brackets.
- ILs do not float.
- Setting an IL on an object has no effect.
- Getting an object's IL will always return ADMIN_LOW.
- Although certain utilities, library functions, and system calls can manipulate IL strings, the resulting ILs are always ADMIN_LOW, and cannot be set on any objects.
- In auditing, the ilabel token is recorded as ADMIN_LOW, when it is recorded. The audit event numbers 519 (AUE_OFLOAT), 520 (AUE_SFLOAT), and 9036 (AUE_iil_change) continue to be reserved, but those events are no longer recorded.

SEE ALSO

Trusted Solaris 8 Reference Manual

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

Trusted Solaris Audit Administration

SunOS 5.8 Reference Manual

attributes(5)

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( ) function specifies the audit log file to the kernel. The kernel writes audit records to this file until an exceptional condition occurs and then the call returns. The *fd* argument is a file descriptor that identifies the audit file. Applications should open this file for writing before calling auditsvc( ).

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

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

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

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

RETURN VALUES

The auditsvc( ) function returns only on an error.

ERRORS

The auditsvc( ) function will fail if:

EAGAIN	The descriptor referred to a *stream*, was marked for System V-style non-blocking I/O, and no data could be written immediately.
EBADF	The *fd* argument is not a valid descriptor open for writing.
EBUSY	A second process attempted to perform this call.
EFBIG	An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.
EINTR	The call is forced to terminate prematurely due to the arrival of a signal whose SV_INTERRUPT bit in sv_flags is set (see sigvec(3UCB)). The signal(3C) function sets this bit for any signal it catches.

EINVAL	Auditing is disabled. See auditon(2). *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	The user's quota of disk blocks on the file system containing the file has been exhausted; audit filesystem space is below the specified limit; or there is no free space remaining on the file system containing the file.
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 4.2 BSD-style non-blocking I/O, and no data could be written immediately.

SUMMARY OF TRUSTED SOLARIS CHANGES

This functionality is active only if auditing is enabled. By default, auditing is enabled in the Trusted Solaris environment. See *Trusted Solaris Audit Administration* for more information.

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

SEE ALSO

Trusted Solaris 8 Reference Manual

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

SunOS 5.8 Reference Manual

sigvec(3UCB)

NAME

chdir, fchdir – Change working directory

SYNOPSIS

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

int fchdir(int fildes);
```

DESCRIPTION

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

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

RETURN VALUES

`chdir( )` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `chdir( )` function will fail if:

`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`	The *path* argument points to an illegal address.
`EINTR`	A signal was caught during the execution of the `chdir( )` function.
`EIO`	An I/O error occurred while reading from or writing to the file system.
`ELOOP`	Too many symbolic links were encountered in translating *path* .
`ENAMETOOLONG`	The length of the *path* argument exceeds `PATH_MAX` , or the length of a *path* component exceeds `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.
`ENOENT`	Either a component of the path prefix or the directory named by *path* does not exist or is a null pathname.

ENOLINK	The *path* argument points to a remote machine and the link to that machine is no longer active.
ENOTDIR	A component of the path name is not a directory.

The `fchdir( )` function will fail if:

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	`chdir( )` is Async-Signal-Safe

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

Trusted Solaris 8 Reference Manual

`chroot`(2)

SunOS 5.8 Reference Manual

`attributes`(5)

NAME

chmod, fchmod – Change access permission mode of file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char * path, mode_t mode);

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

DESCRIPTION

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

`S_ISUID`	04000	Set user ID on execution.
`S_ISGID`	020#0	Set group ID on execution if # is `7` , `5` , `3` , or `1` . Enable mandatory file / record locking if # is `6` , `4` , `2` , or `0` .
`S_ISVTX`	01000	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 the bitwise OR operation of the access permission bits.

The effective user ID of the process must match the owner of the file or the process must have the `PRIV_FILE_SETDAC` 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.

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

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

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

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

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

RETURN VALUES

`chmod( )` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `chmod( )` function will fail if:

`EACCES` Search permission is denied on a component of the path prefix of *path* . 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_DAC_WRITE and/or the PRIV_FILE_MAC_WRITE privilege.
EFAULT	The *path* argument 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* .
ENAMETOOLONG	The length of the *path* argument exceeds PATH_MAX , or the length of a *path* component exceeds NAME_MAX while _POSIX_NO_TRUNC is in effect.
ENOENT	Either a component of the path prefix or the file referred to by *path* does not exist or is a null pathname.
ENOLINK	The *fildes* argument points to a remote machine and the link to that machine is no longer active.
ENOTDIR	A component of the prefix of *path* is not a directory.
EPERM	The effective user ID does not match the owner of the file. 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.

The fchmod( ) function will fail if:

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	chmod( ) is Async-Signal-Safe

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

- To override a search permission error, the calling process requires the PRIV_FILE_MAC_SEARCH privilege.
- To override a write permission error, the calling process requires the PRIV_FILE_DAC_WRITE and/or the PRIV_FILE_MAC_WRITE privilege.
- If the calling process does not own the object, the calling process requires the PRIV_FILE_SETDAC privilege.

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

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

SEE ALSO

Trusted Solaris 8 Reference Manual

chmod(1) , chown(2) , creat(2) , fcntl(2) , mknod(2) , open(2) , read(2) , rename(2) , stat(2) , write(2)

SunOS 5.8 Reference Manual

mkfifo(3C) , attributes(5) , stat(3HEAD)

System Interface Guide

NOTES

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

NAME

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

SYNOPSIS

```
#include <unistd.h>
#include <sys/types.h>
int chown(const char * path, uid_t owner, gid_t group);

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

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

DESCRIPTION

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

The `lchown( )` function sets the owner ID and group ID of the named file in the same manner as `chown( )`, unless the named file is a symbolic link. In this case, `lchown( )` changes the ownership of the symbolic link file itself, while `chown( )` changes the ownership of the file or directory to which the symbolic link refers.

If `chown( )`, `lchown( )`, or `fchown( )` is invoked, the set-user- ID and set-group- ID bits of the file mode, `chmod`(2). respectively, are cleared. See `chmod`(2). 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 not in effect, the effective user ID of the process must match the owner of the file or the process must be the super-user to change the ownership of a file. When `_POSIX_CHOWN_RESTRICTED` is in effect, the `chown( )`, `lchown( )`, and `fchown( )` functions 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.

set rstchown = 1

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

set rstchown = 0

See `system`(4) and `fpathconf`(2).

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

RETURN VALUES

`chown()` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

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

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

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

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

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

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

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

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

`ENOENT` Either a component of the path prefix or the file referred to by *path* does not exist or is a null pathname.

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

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.

The fchown( ) function will fail if:

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	chown( ) is Async-Signal-Safe

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.

SEE ALSO

Trusted Solaris 8 Reference Manual

`chgrp`(1) , `chown`(1) , `chmod`(2)

SunOS 5.8 Reference Manual

`attributes`(5)

NAME

chroot, fchroot – Change root directory

SYNOPSIS

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

int fchroot(int fildes);
```

DESCRIPTION

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

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

The calling process must assert the `PRIV_PROC_CHROOT` privilege to use this system call. While it is always possible to change to the system root using the `fchroot()` function, it is not guaranteed to succeed in any other case, even should *fildes* be valid in all respects.

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

RETURN VALUES

`chroot()` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `chroot()` function will fail if:

`EACCES` Search permission is denied for a component of the path prefix of *dirname* , or search permission is denied for the directory referred to by *dirname* .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` The *path* argument points to an illegal address.

`EINVAL` The `fchroot()` function 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 execution of the `chroot()` function.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *path* .
ENAMETOOLONG	The length of the *path* argument exceeds `PATH_MAX` , or the length of a *path* component exceeds `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.
ENOENT	The named directory does not exist or is a null pathname.
ENOLINK	The *path* argument points to a remote machine and the link to that machine is no longer active.
ENOTDIR	Any component of the path name is not a directory.
EPERM	The 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.

SEE ALSO
Trusted Solaris 8 Reference Manual

`chroot`(1M)

WARNINGS

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

NAME

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

SYNOPSIS

cc [*flags*…] *file*… –ltsol

```
#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

`chstate( )` returns:

`0` On success.

`–1` On failure, 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 fallback 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);
```

DESCRIPTION

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

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

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

- If the group ID of the new file does not match the effective group ID or one of the supplementary group IDs, the `S_ISGID` bit is cleared. The calling process may assert the `PRIV_FILE_SETID` privilege to override clearing of the `S_ISGID` bit.
- All bits set in the process's file mode creation mask (see `umask`(2)) are correspondingly cleared in the file's permission mask.
- The "save text image after execution bit" of the mode is cleared. [See `chmod`(2) for the values of *mode*.] 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.

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

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

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

RETURN VALUES

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

ERRORS

The `creat()` function will fail:

EACCES	Search permission is denied on a component of the path prefix. 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 where the new file entry is being placed cannot be extended because the user's quota of disk blocks on that file system has been exhausted, or the user's quota of inodes on the file system where the file is being created has been exhausted.
EFAULT	The *path* argument points to an illegal address.
EINTR	A signal was caught during the execution of the creat( ) function.
EISDIR	The named file is an existing directory.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMFILE	The process has too many open files. [See getrlimit(2).]
ENFILE	The system file table is full.
ENOENT	A component of the path prefix does not exist, or the path name is null.
ENOLINK	The *path* argument points to a remote machine and the link to that machine is no longer active.
ENOSPC	The file system is out of inodes.
ENOTDIR	A component of the path prefix is not a directory.
EOVERFLOW	The file is a large file at the time of creat( ).

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

USAGE

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

ATTRIBUTES

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

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

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.

Information labels (ILs) are not supported in Trusted Solaris 7 and later releases. Trusted Solaris software interprets any ILs on communications and files from systems running earlier releases as `ADMIN_LOW`.

Objects still have CMW labels, and CMW labels still include the IL component: `IL[SL]`; however, the IL component is fixed at `ADMIN_LOW`.

As a result, Trusted Solaris 7 has the following characteristics:

- ILs do not display in window labels; SLs (Sensitivity Labels) display alone within brackets.
- ILs do not float.
- Setting an IL on an object has no effect.
- Getting an object's IL will always return `ADMIN_LOW`.
- Although certain utilities, library functions, and system calls can manipulate IL strings, the resulting ILs are always `ADMIN_LOW`, and cannot be set on any objects.

SEE ALSO

Trusted Solaris 8 Reference Manual

`chmod`(2), `fcntl`(2), `getrlimit`(2), `lseek`(2), `open`(2), `read`(2), `write`(2)

SunOS 5.8 Reference Manual

`close`(2), `dup`(2), `umask`(2), `attributes`(5), `lf64`(5), `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.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtsu

SEE ALSO

Trusted Solaris 8 Reference Manual

`devpolicy`(1M)

SunOS 5.8 Reference Manual

`attributes`(5)

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, const char * 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

Each of the functions in the `exec` family replace the current process image with a new process image. The new image is constructed from a regular, executable file called the *new process image file* . This file is either an executable object file or a file of data for an interpreter. There is no return from a successful call to one of these functions because the calling process image is overlaid by the new process image.

An interpreter file begins with a line of the form

`#!` *pathname* [*arg*]

where *pathname* is the path of the interpreter, and *arg* is an optional argument. When an interpreter file is executed, the system invokes the specified interpreter. The pathname specified in the interpreter file is passed as *arg0* to the interpreter. If *arg* was specified in the interpreter file, it is passed as *arg1* to the interpreter. The remaining arguments to the interpreter are *arg0* through *argn* of the originally exec'd file. The interpreter named by *pathname* must not be an interpreter file.

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

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

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

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

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

The *file* argument is used to construct a pathname that identifies the new process image file . If the *file* argument contains a slash character, it is used as the pathname for this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed in the PATH environment variable (see environ(5)). The environment is supplied typically by the shell. If the process image file is not a valid executable object file, execlp( ) and execvp( ) use the contents of that file as standard input to the shell. In this case, the shell becomes the new process image. In a standard-conforming application (see standards(5)), the exec family of functions use /usr/bin/ksh (see ksh(1)); otherwise, they use /usr/bin/sh (see sh(1)).

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

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

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

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

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
```

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

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

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

setlocale(LC_ALL, "C")

is executed at startup.

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

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

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

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

If the 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( ) .

Any shared memory segments attached to the calling process image will not be attached to the new process image (see shmop(2)). Any mappings established through mmap( ) are not preserved across an exec . Memory mappings created

in the process are unmapped before the address space is rebuilt for the new process image. (see mmap(2)).

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

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

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

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

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

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

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

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 ID s
- semadj values (see semop(2))
- session membership (see exit(2) and signal(3C))
- real user ID
- real group ID
- trace flag (see ptrace(2) request 0)
- time left until an alarm clock signal (see alarm(2))

- current working directory
- root directory
- file mode creation mask (see umask(2))
- file size limit (see ulimit(2))
- resource limits (see getrlimit(2))
- tms_utime , tms_stime , tms_cutime , and tms_cstime (see times(2))
- file-locks (see fcntl(2) and lockf(3C))
- controlling terminal
- process signal mask (see sigprocmask(2))
- pending signals (see sigpending(2))
- clearance (see getclearance(2))
- sensitivity label (see getcmwlabel(2))
- inheritable privilege set (see getppriv(2))
- process attribute flags (see getpattr(2))

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

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

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

RETURN VALUES

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

ERRORS

The `exec` functions will fail if:

`E2BIG` The number of bytes in the new process's argument list is greater than the system-imposed limit of `ARG_MAX` bytes. The argument list limit is sum of the size of the argument list plus the size of the environment's exported shell variables.

`EACCES` Search permission is denied for a directory listed in the new process file's path prefix; the new process file is not an ordinary file; or the new process file mode denies execute permission. Moreover, the calling process does not have `PRIV_FILE_DAC_SEARCH` and/or `PRIV_FILE_MAC_SEARCH` to override the restriction.

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

`EFAULT` An argument points to an illegal address.

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

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

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

ENOENT One or more components of the new process path name of the file do not exist or is a null pathname.

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

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

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

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

The exec functions may fail if:

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

ENOMEM The new process image requires more memory than RLIMIT_VMEM , the limit imposed by setrlimit( ) (see brk(2)).

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

USAGE

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

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

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	execle( ) and execve( ) are Async-Signal-Safe

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.

Information labels (IL s) are not supported in Trusted Solaris 7 and later releases. Trusted Solaris software interprets any IL s on communications and files from systems running earlier releases as ADMIN_LOW .

Objects still have CMW labels, and CMW labels still include the IL component: IL[SL] ; however, the IL component is fixed at ADMIN_LOW .

As a result, Trusted Solaris 7 and later versions have the following characteristics:

- IL s do not display in window labels; SL s (Sensitivity Labels) display alone within brackets.
- IL s do not float.
- Setting an IL on an object has no effect.
- Getting an object's IL will always return ADMIN_LOW .
- Although certain utilities, library functions, and system calls can manipulate IL strings, the resulting IL s are always ADMIN_LOW , and cannot be set on any objects.

SEE ALSO

Trusted Solaris 8 Reference Manual

chmod(2) , fcntl(2) , fork(2) , getrlimit(2) , nice(2) , priocntl(2) , semop(2) , shmop(2) , setlocale(3C) ,

SunOS 5.8 Reference Manual

ksh(1) , ps(1) , sh(1) , alarm(2) , brk(2) , exit(2) , mmap(2) , profil(2) , ptrace(2) , sigpending(2) , sigprocmask(2) , times(2) , umask(2) , lockf(3C) , signal(3C) , system(3C) , timer_create(3RT) , a.out(4) , attributes(5) , environ(5) , standards(5)

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, const char * 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

Each of the functions in the `exec` family replace the current process image with a new process image. The new image is constructed from a regular, executable file called the *new process image file* . This file is either an executable object file or a file of data for an interpreter. There is no return from a successful call to one of these functions because the calling process image is overlaid by the new process image.

An interpreter file begins with a line of the form

`#!` *pathname* [*arg*]

where *pathname* is the path of the interpreter, and *arg* is an optional argument. When an interpreter file is executed, the system invokes the specified interpreter. The pathname specified in the interpreter file is passed as *arg0* to the interpreter. If *arg* was specified in the interpreter file, it is passed as *arg1* to the interpreter. The remaining arguments to the interpreter are *arg0* through *argn* of the originally exec'd file. The interpreter named by *pathname* must not be an interpreter file.

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

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

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

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

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

The *file* argument is used to construct a pathname that identifies the new process image file . If the *file* argument contains a slash character, it is used as the pathname for this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed in the PATH environment variable (see environ(5)). The environment is supplied typically by the shell. If the process image file is not a valid executable object file, execlp( ) and execvp( ) use the contents of that file as standard input to the shell. In this case, the shell becomes the new process image. In a standard-conforming application (see standards(5)), the exec family of functions use /usr/bin/ksh (see ksh(1)); otherwise, they use /usr/bin/sh (see sh(1)).

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

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

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

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

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
```

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

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

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

setlocale(LC_ALL, "C")

is executed at startup.

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

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

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

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

If the 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( ) .

Any shared memory segments attached to the calling process image will not be attached to the new process image (see shmop(2)). Any mappings established through mmap( ) are not preserved across an exec . Memory mappings created

in the process are unmapped before the address space is rebuilt for the new process image. (see mmap(2)).

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

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

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

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

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

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

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

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 ID s
- semadj values (see semop(2))
- session membership (see exit(2) and signal(3C))
- real user ID
- real group ID
- trace flag (see ptrace(2) request 0)
- time left until an alarm clock signal (see alarm(2))

- current working directory
- root directory
- file mode creation mask (see `umask`(2))
- file size limit (see `ulimit`(2))
- resource limits (see `getrlimit`(2))
- `tms_utime` , `tms_stime` , `tms_cutime` , and `tms_cstime` (see `times`(2))
- file-locks (see `fcntl`(2) and `lockf`(3C))
- controlling terminal
- process signal mask (see `sigprocmask`(2))
- pending signals (see `sigpending`(2))
- clearance (see `getclearance`(2))
- sensitivity label (see `getcmwlabel`(2))
- inheritable privilege set (see `getppriv`(2))
- process attribute flags (see `getpattr`(2))

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

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

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

RETURN VALUES

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

ERRORS

The exec functions will fail if:

E2BIG — The number of bytes in the new process's argument list is greater than the system-imposed limit of ARG_MAX bytes. The argument list limit is sum of the size of the argument list plus the size of the environment's exported shell variables.

EACCES — Search permission is denied for a directory listed in the new process file's path prefix; the new process file is not an ordinary file; or the new process file mode denies execute permission. Moreover, the calling process does not have PRIV_FILE_DAC_SEARCH and/or PRIV_FILE_MAC_SEARCH to override the restriction.

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

EFAULT — An argument points to an illegal address.

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

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

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

ENOENT One or more components of the new process path name of the file do not exist or is a null pathname.

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

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

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

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

The exec functions may fail if:

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

ENOMEM The new process image requires more memory than RLIMIT_VMEM , the limit imposed by setrlimit( ) (see brk(2)).

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

USAGE

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

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

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	execle( ) and execve( ) are Async-Signal-Safe

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.

Information labels (IL s) are not supported in Trusted Solaris 7 and later releases. Trusted Solaris software interprets any IL s on communications and files from systems running earlier releases as ADMIN_LOW .

Objects still have CMW labels, and CMW labels still include the IL component: IL[SL] ; however, the IL component is fixed at ADMIN_LOW .

As a result, Trusted Solaris 7 and later versions have the following characteristics:

- IL s do not display in window labels; SL s (Sensitivity Labels) display alone within brackets.
- IL s do not float.
- Setting an IL on an object has no effect.
- Getting an object's IL will always return ADMIN_LOW .
- Although certain utilities, library functions, and system calls can manipulate IL strings, the resulting IL s are always ADMIN_LOW , and cannot be set on any objects.

SEE ALSO

Trusted Solaris 8 Reference Manual

chmod(2) , fcntl(2) , fork(2) , getrlimit(2) , nice(2) , priocntl(2) , semop(2) , shmop(2) , setlocale(3C) ,

SunOS 5.8 Reference Manual

ksh(1) , ps(1) , sh(1) , alarm(2) , brk(2) , exit(2) , mmap(2) , profil(2) , ptrace(2) , sigpending(2) , sigprocmask(2) , times(2) , umask(2) , lockf(3C) , signal(3C) , system(3C) , timer_create(3RT) , a.out(4) , attributes(5) , environ(5) , standards(5)

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, const char * 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

Each of the functions in the `exec` family replace the current process image with a new process image. The new image is constructed from a regular, executable file called the *new process image file* . This file is either an executable object file or a file of data for an interpreter. There is no return from a successful call to one of these functions because the calling process image is overlaid by the new process image.

An interpreter file begins with a line of the form

`#!` *pathname* [*arg*]

where *pathname* is the path of the interpreter, and *arg* is an optional argument. When an interpreter file is executed, the system invokes the specified interpreter. The pathname specified in the interpreter file is passed as *arg0* to the interpreter. If *arg* was specified in the interpreter file, it is passed as *arg1* to the interpreter. The remaining arguments to the interpreter are *arg0* through *argn* of the originally exec'd file. The interpreter named by *pathname* must not be an interpreter file.

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

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

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

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

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

The *file* argument is used to construct a pathname that identifies the new process image file . If the *file* argument contains a slash character, it is used as the pathname for this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed in the PATH environment variable (see environ(5)). The environment is supplied typically by the shell. If the process image file is not a valid executable object file, execlp( ) and execvp( ) use the contents of that file as standard input to the shell. In this case, the shell becomes the new process image. In a standard-conforming application (see standards(5)), the exec family of functions use /usr/bin/ksh (see ksh(1)); otherwise, they use /usr/bin/sh (see sh(1)).

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

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

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

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

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
```

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

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

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

setlocale(LC_ALL, "C")

is executed at startup.

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

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

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

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

If the 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( ) .

Any shared memory segments attached to the calling process image will not be attached to the new process image (see shmop(2)). Any mappings established through mmap( ) are not preserved across an exec . Memory mappings created

in the process are unmapped before the address space is rebuilt for the new process image. (see mmap(2)).

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

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

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

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

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

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

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

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 ID s
- semadj values (see semop(2))
- session membership (see exit(2) and signal(3C))
- real user ID
- real group ID
- trace flag (see ptrace(2) request 0)
- time left until an alarm clock signal (see alarm(2))

- current working directory
- root directory
- file mode creation mask (see umask(2))
- file size limit (see ulimit(2))
- resource limits (see getrlimit(2))
- tms_utime , tms_stime , tms_cutime , and tms_cstime (see times(2))
- file-locks (see fcntl(2) and lockf(3C))
- controlling terminal
- process signal mask (see sigprocmask(2))
- pending signals (see sigpending(2))
- clearance (see getclearance(2))
- sensitivity label (see getcmwlabel(2))
- inheritable privilege set (see getppriv(2))
- process attribute flags (see getpattr(2))

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

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

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

RETURN VALUES

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

ERRORS

The exec functions will fail if:

E2BIG — The number of bytes in the new process's argument list is greater than the system-imposed limit of ARG_MAX bytes. The argument list limit is sum of the size of the argument list plus the size of the environment's exported shell variables.

EACCES — Search permission is denied for a directory listed in the new process file's path prefix; the new process file is not an ordinary file; or the new process file mode denies execute permission. Moreover, the calling process does not have PRIV_FILE_DAC_SEARCH and/or PRIV_FILE_MAC_SEARCH to override the restriction.

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

EFAULT — An argument points to an illegal address.

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

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

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

ENOENT One or more components of the new process path name of the file do not exist or is a null pathname.

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

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

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

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

The exec functions may fail if:

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

ENOMEM The new process image requires more memory than RLIMIT_VMEM , the limit imposed by setrlimit( ) (see brk(2)).

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

USAGE

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

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

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	execle( ) and execve( ) are Async-Signal-Safe

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.

Information labels (IL s) are not supported in Trusted Solaris 7 and later releases. Trusted Solaris software interprets any IL s on communications and files from systems running earlier releases as ADMIN_LOW .

Objects still have CMW labels, and CMW labels still include the IL component: IL[SL] ; however, the IL component is fixed at ADMIN_LOW .

As a result, Trusted Solaris 7 and later versions have the following characteristics:

- IL s do not display in window labels; SL s (Sensitivity Labels) display alone within brackets.
- IL s do not float.
- Setting an IL on an object has no effect.
- Getting an object's IL will always return ADMIN_LOW .
- Although certain utilities, library functions, and system calls can manipulate IL strings, the resulting IL s are always ADMIN_LOW , and cannot be set on any objects.

SEE ALSO

Trusted Solaris 8 Reference Manual

chmod(2) , fcntl(2) , fork(2) , getrlimit(2) , nice(2) , priocntl(2) , semop(2) , shmop(2) , setlocale(3C) ,

SunOS 5.8 Reference Manual

ksh(1) , ps(1) , sh(1) , alarm(2) , brk(2) , exit(2) , mmap(2) , profil(2) , ptrace(2) , sigpending(2) , sigprocmask(2) , times(2) , umask(2) , lockf(3C) , signal(3C) , system(3C) , timer_create(3RT) , a.out(4) , attributes(5) , environ(5) , standards(5)

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, const char * 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

Each of the functions in the `exec` family replace the current process image with a new process image. The new image is constructed from a regular, executable file called the *new process image file* . This file is either an executable object file or a file of data for an interpreter. There is no return from a successful call to one of these functions because the calling process image is overlaid by the new process image.

An interpreter file begins with a line of the form

`#!` *pathname* [*arg*]

where *pathname* is the path of the interpreter, and *arg* is an optional argument. When an interpreter file is executed, the system invokes the specified interpreter. The pathname specified in the interpreter file is passed as *arg0* to the interpreter. If *arg* was specified in the interpreter file, it is passed as *arg1* to the interpreter. The remaining arguments to the interpreter are *arg0* through *argn* of the originally exec'd file. The interpreter named by *pathname* must not be an interpreter file.

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

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

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

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

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

The *file* argument is used to construct a pathname that identifies the new process image file . If the *file* argument contains a slash character, it is used as the pathname for this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed in the PATH environment variable (see environ(5)). The environment is supplied typically by the shell. If the process image file is not a valid executable object file, execlp( ) and execvp( ) use the contents of that file as standard input to the shell. In this case, the shell becomes the new process image. In a standard-conforming application (see standards(5)), the exec family of functions use /usr/bin/ksh (see ksh(1)); otherwise, they use /usr/bin/sh (see sh(1)).

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

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

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

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

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
```

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

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

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

setlocale(LC_ALL, "C")

is executed at startup.

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

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

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

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

If the 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( ) .

Any shared memory segments attached to the calling process image will not be attached to the new process image (see shmop(2)). Any mappings established through mmap( ) are not preserved across an exec . Memory mappings created

in the process are unmapped before the address space is rebuilt for the new process image. (see mmap(2)).

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

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

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

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

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

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

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

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 ID s
- semadj values (see semop(2))
- session membership (see exit(2) and signal(3C))
- real user ID
- real group ID
- trace flag (see ptrace(2) request 0)
- time left until an alarm clock signal (see alarm(2))

- current working directory
- root directory
- file mode creation mask (see umask(2))
- file size limit (see ulimit(2))
- resource limits (see getrlimit(2))
- tms_utime , tms_stime , tms_cutime , and tms_cstime (see times(2))
- file-locks (see fcntl(2) and lockf(3C))
- controlling terminal
- process signal mask (see sigprocmask(2))
- pending signals (see sigpending(2))
- clearance (see getclearance(2))
- sensitivity label (see getcmwlabel(2))
- inheritable privilege set (see getppriv(2))
- process attribute flags (see getpattr(2))

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

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

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

RETURN VALUES

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

ERRORS

The exec functions will fail if:

E2BIG — The number of bytes in the new process's argument list is greater than the system-imposed limit of ARG_MAX bytes. The argument list limit is sum of the size of the argument list plus the size of the environment's exported shell variables.

EACCES — Search permission is denied for a directory listed in the new process file's path prefix; the new process file is not an ordinary file; or the new process file mode denies execute permission. Moreover, the calling process does not have PRIV_FILE_DAC_SEARCH and/or PRIV_FILE_MAC_SEARCH to override the restriction.

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

EFAULT — An argument points to an illegal address.

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

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

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

ENOENT One or more components of the new process path name of the file do not exist or is a null pathname.

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

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

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

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

The exec functions may fail if:

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

ENOMEM The new process image requires more memory than RLIMIT_VMEM , the limit imposed by setrlimit( ) (see brk(2)).

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

USAGE

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

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

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	execle( ) and execve( ) are Async-Signal-Safe

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.

Information labels (IL s) are not supported in Trusted Solaris 7 and later releases. Trusted Solaris software interprets any IL s on communications and files from systems running earlier releases as ADMIN_LOW .

Objects still have CMW labels, and CMW labels still include the IL component: IL[SL] ; however, the IL component is fixed at ADMIN_LOW .

As a result, Trusted Solaris 7 and later versions have the following characteristics:

- IL s do not display in window labels; SL s (Sensitivity Labels) display alone within brackets.
- IL s do not float.
- Setting an IL on an object has no effect.
- Getting an object's IL will always return ADMIN_LOW .
- Although certain utilities, library functions, and system calls can manipulate IL strings, the resulting IL s are always ADMIN_LOW , and cannot be set on any objects.

SEE ALSO

Trusted Solaris 8 Reference Manual

chmod(2) , fcntl(2) , fork(2) , getrlimit(2) , nice(2) , priocntl(2) , semop(2) , shmop(2) , setlocale(3C) ,

SunOS 5.8 Reference Manual

ksh(1) , ps(1) , sh(1) , alarm(2) , brk(2) , exit(2) , mmap(2) , profil(2) , ptrace(2) , sigpending(2) , sigprocmask(2) , times(2) , umask(2) , lockf(3C) , signal(3C) , system(3C) , timer_create(3RT) , a.out(4) , attributes(5) , environ(5) , standards(5)

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, const char * 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

Each of the functions in the `exec` family replace the current process image with a new process image. The new image is constructed from a regular, executable file called the *new process image file* . This file is either an executable object file or a file of data for an interpreter. There is no return from a successful call to one of these functions because the calling process image is overlaid by the new process image.

An interpreter file begins with a line of the form

`#!` *pathname* [*arg*]

where *pathname* is the path of the interpreter, and *arg* is an optional argument. When an interpreter file is executed, the system invokes the specified interpreter. The pathname specified in the interpreter file is passed as *arg0* to the interpreter. If *arg* was specified in the interpreter file, it is passed as *arg1* to the interpreter. The remaining arguments to the interpreter are *arg0* through *argn* of the originally exec'd file. The interpreter named by *pathname* must not be an interpreter file.

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

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

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

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

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

The *file* argument is used to construct a pathname that identifies the new process image file . If the *file* argument contains a slash character, it is used as the pathname for this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed in the PATH environment variable (see environ(5)). The environment is supplied typically by the shell. If the process image file is not a valid executable object file, execlp( ) and execvp( ) use the contents of that file as standard input to the shell. In this case, the shell becomes the new process image. In a standard-conforming application (see standards(5)), the exec family of functions use /usr/bin/ksh (see ksh(1)); otherwise, they use /usr/bin/sh (see sh(1)).

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

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

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

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

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
```

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

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

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

setlocale(LC_ALL, "C")

is executed at startup.

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

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

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

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

If the 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( ) .

Any shared memory segments attached to the calling process image will not be attached to the new process image (see shmop(2)). Any mappings established through mmap( ) are not preserved across an exec . Memory mappings created

in the process are unmapped before the address space is rebuilt for the new process image. (see mmap(2)).

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

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

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

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

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

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

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

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 ID s
- semadj values (see semop(2))
- session membership (see exit(2) and signal(3C))
- real user ID
- real group ID
- trace flag (see ptrace(2) request 0)
- time left until an alarm clock signal (see alarm(2))

- current working directory
- root directory
- file mode creation mask (see `umask`(2))
- file size limit (see `ulimit`(2))
- resource limits (see `getrlimit`(2))
- `tms_utime` , `tms_stime` , `tms_cutime` , and `tms_cstime` (see `times`(2))
- file-locks (see `fcntl`(2) and `lockf`(3C))
- controlling terminal
- process signal mask (see `sigprocmask`(2))
- pending signals (see `sigpending`(2))
- clearance (see `getclearance`(2))
- sensitivity label (see `getcmwlabel`(2))
- inheritable privilege set (see `getppriv`(2))
- process attribute flags (see `getpattr`(2))

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

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

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

RETURN VALUES

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

ERRORS

The `exec` functions will fail if:

`E2BIG` The number of bytes in the new process's argument list is greater than the system-imposed limit of `ARG_MAX` bytes. The argument list limit is sum of the size of the argument list plus the size of the environment's exported shell variables.

`EACCES` Search permission is denied for a directory listed in the new process file's path prefix; the new process file is not an ordinary file; or the new process file mode denies execute permission. Moreover, the calling process does not have `PRIV_FILE_DAC_SEARCH` and/or `PRIV_FILE_MAC_SEARCH` to override the restriction.

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

`EFAULT` An argument points to an illegal address.

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

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

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

ENOENT One or more components of the new process path name of the file do not exist or is a null pathname.

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

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

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

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

The exec functions may fail if:

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

ENOMEM The new process image requires more memory than RLIMIT_VMEM , the limit imposed by setrlimit( ) (see brk(2)).

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

USAGE

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

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

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	execle( ) and execve( ) are Async-Signal-Safe

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.

Information labels (IL s) are not supported in Trusted Solaris 7 and later releases. Trusted Solaris software interprets any IL s on communications and files from systems running earlier releases as ADMIN_LOW .

Objects still have CMW labels, and CMW labels still include the IL component: IL[SL] ; however, the IL component is fixed at ADMIN_LOW .

As a result, Trusted Solaris 7 and later versions have the following characteristics:

- IL s do not display in window labels; SL s (Sensitivity Labels) display alone within brackets.
- IL s do not float.
- Setting an IL on an object has no effect.
- Getting an object's IL will always return ADMIN_LOW .
- Although certain utilities, library functions, and system calls can manipulate IL strings, the resulting IL s are always ADMIN_LOW , and cannot be set on any objects.

SEE ALSO

Trusted Solaris 8 Reference Manual

chmod(2) , fcntl(2) , fork(2) , getrlimit(2) , nice(2) , priocntl(2) , semop(2) , shmop(2) , setlocale(3C) ,

SunOS 5.8 Reference Manual

ksh(1) , ps(1) , sh(1) , alarm(2) , brk(2) , exit(2) , mmap(2) , profil(2) , ptrace(2) , sigpending(2) , sigprocmask(2) , times(2) , umask(2) , lockf(3C) , signal(3C) , system(3C) , timer_create(3RT) , a.out(4) , attributes(5) , environ(5) , standards(5)

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, const char * 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

Each of the functions in the `exec` family replace the current process image with a new process image. The new image is constructed from a regular, executable file called the *new process image file* . This file is either an executable object file or a file of data for an interpreter. There is no return from a successful call to one of these functions because the calling process image is overlaid by the new process image.

An interpreter file begins with a line of the form

`#!` *pathname* [*arg*]

where *pathname* is the path of the interpreter, and *arg* is an optional argument. When an interpreter file is executed, the system invokes the specified interpreter. The pathname specified in the interpreter file is passed as *arg0* to the interpreter. If *arg* was specified in the interpreter file, it is passed as *arg1* to the interpreter. The remaining arguments to the interpreter are *arg0* through *argn* of the originally exec'd file. The interpreter named by *pathname* must not be an interpreter file.

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

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

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

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

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

The *file* argument is used to construct a pathname that identifies the new process image file . If the *file* argument contains a slash character, it is used as the pathname for this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed in the PATH environment variable (see environ(5)). The environment is supplied typically by the shell. If the process image file is not a valid executable object file, execlp( ) and execvp( ) use the contents of that file as standard input to the shell. In this case, the shell becomes the new process image. In a standard-conforming application (see standards(5)), the exec family of functions use /usr/bin/ksh (see ksh(1)); otherwise, they use /usr/bin/sh (see sh(1)).

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

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

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

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

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
```

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

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

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

setlocale(LC_ALL, "C")

is executed at startup.

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

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

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

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

If the 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( ) .

Any shared memory segments attached to the calling process image will not be attached to the new process image (see shmop(2)). Any mappings established through mmap( ) are not preserved across an exec . Memory mappings created

in the process are unmapped before the address space is rebuilt for the new process image. (see mmap(2)).

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

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

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

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

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

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

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

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 ID s
- semadj values (see semop(2))
- session membership (see exit(2) and signal(3C))
- real user ID
- real group ID
- trace flag (see ptrace(2) request 0)
- time left until an alarm clock signal (see alarm(2))

- current working directory
- root directory
- file mode creation mask (see `umask`(2))
- file size limit (see `ulimit`(2))
- resource limits (see `getrlimit`(2))
- `tms_utime` , `tms_stime` , `tms_cutime` , and `tms_cstime` (see `times`(2))
- file-locks (see `fcntl`(2) and `lockf`(3C))
- controlling terminal
- process signal mask (see `sigprocmask`(2))
- pending signals (see `sigpending`(2))
- clearance (see `getclearance`(2))
- sensitivity label (see `getcmwlabel`(2))
- inheritable privilege set (see `getppriv`(2))
- process attribute flags (see `getpattr`(2))

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

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

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

RETURN VALUES

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

ERRORS

The `exec` functions will fail if:

`E2BIG` The number of bytes in the new process's argument list is greater than the system-imposed limit of `ARG_MAX` bytes. The argument list limit is sum of the size of the argument list plus the size of the environment's exported shell variables.

`EACCES` Search permission is denied for a directory listed in the new process file's path prefix; the new process file is not an ordinary file; or the new process file mode denies execute permission. Moreover, the calling process does not have `PRIV_FILE_DAC_SEARCH` and/or `PRIV_FILE_MAC_SEARCH` to override the restriction.

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

`EFAULT` An argument points to an illegal address.

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

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

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

ENOENT One or more components of the new process path name of the file do not exist or is a null pathname.

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

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

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

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

The exec functions may fail if:

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

ENOMEM The new process image requires more memory than RLIMIT_VMEM , the limit imposed by setrlimit( ) (see brk(2)).

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

USAGE

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

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

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	execle( ) and execve( ) are Async-Signal-Safe

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.

Information labels (IL s) are not supported in Trusted Solaris 7 and later releases. Trusted Solaris software interprets any IL s on communications and files from systems running earlier releases as ADMIN_LOW .

Objects still have CMW labels, and CMW labels still include the IL component: IL[SL] ; however, the IL component is fixed at ADMIN_LOW .

As a result, Trusted Solaris 7 and later versions have the following characteristics:

- IL s do not display in window labels; SL s (Sensitivity Labels) display alone within brackets.
- IL s do not float.
- Setting an IL on an object has no effect.
- Getting an object's IL will always return ADMIN_LOW .
- Although certain utilities, library functions, and system calls can manipulate IL strings, the resulting IL s are always ADMIN_LOW , and cannot be set on any objects.

SEE ALSO

Trusted Solaris 8 Reference Manual

chmod(2) , fcntl(2) , fork(2) , getrlimit(2) , nice(2) , priocntl(2) , semop(2) , shmop(2) , setlocale(3C) ,

SunOS 5.8 Reference Manual

ksh(1) , ps(1) , sh(1) , alarm(2) , brk(2) , exit(2) , mmap(2) , profil(2) , ptrace(2) , sigpending(2) , sigprocmask(2) , times(2) , umask(2) , lockf(3C) , signal(3C) , system(3C) , timer_create(3RT) , a.out(4) , attributes(5) , environ(5) , standards(5)

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, const char * 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

Each of the functions in the `exec` family replace the current process image with a new process image. The new image is constructed from a regular, executable file called the *new process image file* . This file is either an executable object file or a file of data for an interpreter. There is no return from a successful call to one of these functions because the calling process image is overlaid by the new process image.

An interpreter file begins with a line of the form

`#!` *pathname* [*arg*]

where *pathname* is the path of the interpreter, and *arg* is an optional argument. When an interpreter file is executed, the system invokes the specified interpreter. The pathname specified in the interpreter file is passed as *arg0* to the interpreter. If *arg* was specified in the interpreter file, it is passed as *arg1* to the interpreter. The remaining arguments to the interpreter are *arg0* through *argn* of the originally exec'd file. The interpreter named by *pathname* must not be an interpreter file.

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

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

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

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

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

The *file* argument is used to construct a pathname that identifies the new process image file . If the *file* argument contains a slash character, it is used as the pathname for this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed in the PATH environment variable (see environ(5)). The environment is supplied typically by the shell. If the process image file is not a valid executable object file, execlp( ) and execvp( ) use the contents of that file as standard input to the shell. In this case, the shell becomes the new process image. In a standard-conforming application (see standards(5)), the exec family of functions use /usr/bin/ksh (see ksh(1)); otherwise, they use /usr/bin/sh (see sh(1)).

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

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

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

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

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
```

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

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

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

setlocale(LC_ALL, "C")

is executed at startup.

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

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

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

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

If the 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( ) .

Any shared memory segments attached to the calling process image will not be attached to the new process image (see shmop(2)). Any mappings established through mmap( ) are not preserved across an exec . Memory mappings created

in the process are unmapped before the address space is rebuilt for the new process image. (see mmap(2)).

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

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

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

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

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

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

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

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 ID s
- semadj values (see semop(2))
- session membership (see exit(2) and signal(3C))
- real user ID
- real group ID
- trace flag (see ptrace(2) request 0)
- time left until an alarm clock signal (see alarm(2))

- current working directory
- root directory
- file mode creation mask (see `umask`(2))
- file size limit (see `ulimit`(2))
- resource limits (see `getrlimit`(2))
- `tms_utime` , `tms_stime` , `tms_cutime` , and `tms_cstime` (see `times`(2))
- file-locks (see `fcntl`(2) and `lockf`(3C))
- controlling terminal
- process signal mask (see `sigprocmask`(2))
- pending signals (see `sigpending`(2))
- clearance (see `getclearance`(2))
- sensitivity label (see `getcmwlabel`(2))
- inheritable privilege set (see `getppriv`(2))
- process attribute flags (see `getpattr`(2))

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

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

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

RETURN VALUES

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

ERRORS

The `exec` functions will fail if:

`E2BIG` The number of bytes in the new process's argument list is greater than the system-imposed limit of `ARG_MAX` bytes. The argument list limit is sum of the size of the argument list plus the size of the environment's exported shell variables.

`EACCES` Search permission is denied for a directory listed in the new process file's path prefix; the new process file is not an ordinary file; or the new process file mode denies execute permission. Moreover, the calling process does not have `PRIV_FILE_DAC_SEARCH` and/or `PRIV_FILE_MAC_SEARCH` to override the restriction.

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

`EFAULT` An argument points to an illegal address.

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

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

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

ENOENT One or more components of the new process path name of the file do not exist or is a null pathname.

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

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

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

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

The exec functions may fail if:

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

ENOMEM The new process image requires more memory than RLIMIT_VMEM , the limit imposed by setrlimit( ) (see brk(2)).

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

USAGE

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

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

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	execle( ) and execve( ) are Async-Signal-Safe

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.

Information labels (IL s) are not supported in Trusted Solaris 7 and later releases. Trusted Solaris software interprets any IL s on communications and files from systems running earlier releases as ADMIN_LOW .

Objects still have CMW labels, and CMW labels still include the IL component: IL[SL] ; however, the IL component is fixed at ADMIN_LOW .

As a result, Trusted Solaris 7 and later versions have the following characteristics:

- IL s do not display in window labels; SL s (Sensitivity Labels) display alone within brackets.
- IL s do not float.
- Setting an IL on an object has no effect.
- Getting an object's IL will always return ADMIN_LOW .
- Although certain utilities, library functions, and system calls can manipulate IL strings, the resulting IL s are always ADMIN_LOW , and cannot be set on any objects.

SEE ALSO

Trusted Solaris 8 Reference Manual

chmod(2) , fcntl(2) , fork(2) , getrlimit(2) , nice(2) , priocntl(2) , semop(2) , shmop(2) , setlocale(3C) ,

SunOS 5.8 Reference Manual

ksh(1) , ps(1) , sh(1) , alarm(2) , brk(2) , exit(2) , mmap(2) , profil(2) , ptrace(2) , sigpending(2) , sigprocmask(2) , times(2) , umask(2) , lockf(3C) , signal(3C) , system(3C) , timer_create(3RT) , a.out(4) , attributes(5) , environ(5) , standards(5)

NAME

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

SYNOPSIS

```
#include <sys/acl.h>
int acl(char * pathp, int cmd, int nentries, aclent_t * aclbufp);

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

DESCRIPTION

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

The following values for *cmd* are supported:

`SETACL` *nentries* ACL entries, specified in buffer *aclbufp* , are stored in the file's ACL . This command can 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.

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

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

RETURN VALUES

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

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

ERRORS

The `acl()` function will fail if:

`EACCESS` The caller does not have 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` .

The *cmd* argument is `SETACL` and *nentries* is less than three.

The *cmd* argument is `SETACL` and the ACL specified in *aclbufp* is not valid.

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

`EINVAL` The *cmd* argument is not `GETACL` , `SETACL` , or `GETACLCNT` ; the *cmd* argument is `SETACL` and *nentries* is less than 3;

or the *cmd* argument is SETACL and the ACL specified in *aclbufp* is not valid.

EIO — A disk I/O error has occurred while storing or retrieving the ACL .

ENOENT — A component of the path does not exist.

ENOSPC — The *cmd* argument is GETACL and *nentries* is less than the number of entries in the file's ACL , or the *cmd* argument is SETACL and there is insufficient space in the file system to store the ACL .

ENOTDIR — A component of the path specified by *pathp* is not a directory, or the *cmd* argument is SETACL and an attempt is made to set a default ACL on a file type other than a directory.

ENOSYS — The *cmd* argument is SETACL and the file specified by *pathp* resides on a file system that does not support ACLs, or the acl( ) function is not supported by this implementation.

EPERM — The *cmd* argument is SETACL and the effective user ID of the caller does not match the owner of the file. To override this restriction, the calling process may assert the PRIV_FILE_SETDAC privilege.

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access or ownership checks.

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

SEE ALSO

SunOS 5.8 Reference Manual

getfacl(1), setfacl(1), aclcheck(3SEC), aclsort(3SEC), attributes(5)

NAME

chdir, fchdir – Change working directory

SYNOPSIS

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

int fchdir(int fildes);
```

DESCRIPTION

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

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

RETURN VALUES

`chdir( )` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `chdir( )` function will fail if:

`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`	The *path* argument points to an illegal address.
`EINTR`	A signal was caught during the execution of the `chdir( )` function.
`EIO`	An I/O error occurred while reading from or writing to the file system.
`ELOOP`	Too many symbolic links were encountered in translating *path* .
`ENAMETOOLONG`	The length of the *path* argument exceeds `PATH_MAX` , or the length of a *path* component exceeds `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.
`ENOENT`	Either a component of the path prefix or the directory named by *path* does not exist or is a null pathname.

ENOLINK	The *path* argument points to a remote machine and the link to that machine is no longer active.
ENOTDIR	A component of the path name is not a directory.

The `fchdir( )` function will fail if:

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	`chdir( )` is Async-Signal-Safe

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

Trusted Solaris 8 Reference Manual

`chroot`(2)

SunOS 5.8 Reference Manual

`attributes`(5)

NAME

chmod, fchmod – Change access permission mode of file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char * path, mode_t mode);

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

DESCRIPTION

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

`S_ISUID`	04000	Set user ID on execution.
`S_ISGID`	020#0	Set group ID on execution if # is `7` , `5` , `3` , or `1` . Enable mandatory file / record locking if # is `6` , `4` , `2` , or `0` .
`S_ISVTX`	01000	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 the bitwise OR operation of the access permission bits.

The effective user ID of the process must match the owner of the file or the process must have the `PRIV_FILE_SETDAC` 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.

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

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

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

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

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

RETURN VALUES

chmod( ) returns:

0 On success.

−1 On failure, and sets errno to indicate the error.

ERRORS

The chmod( ) function will fail if:

EACCES Search permission is denied on a component of the path prefix of *path* . 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_DAC_WRITE and/or the PRIV_FILE_MAC_WRITE privilege.
EFAULT	The *path* argument 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* .
ENAMETOOLONG	The length of the *path* argument exceeds PATH_MAX , or the length of a *path* component exceeds NAME_MAX while _POSIX_NO_TRUNC is in effect.
ENOENT	Either a component of the path prefix or the file referred to by *path* does not exist or is a null pathname.
ENOLINK	The *fildes* argument points to a remote machine and the link to that machine is no longer active.
ENOTDIR	A component of the prefix of *path* is not a directory.
EPERM	The effective user ID does not match the owner of the file. 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.

The fchmod( ) function will fail if:

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	chmod( ) is Async-Signal-Safe

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

- To override a search permission error, the calling process requires the PRIV_FILE_MAC_SEARCH privilege.
- To override a write permission error, the calling process requires the PRIV_FILE_DAC_WRITE and/or the PRIV_FILE_MAC_WRITE privilege.
- If the calling process does not own the object, the calling process requires the PRIV_FILE_SETDAC privilege.

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

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

SEE ALSO

Trusted Solaris 8 Reference Manual

chmod(1) , chown(2) , creat(2) , fcntl(2) , mknod(2) , open(2) , read(2) , rename(2) , stat(2) , write(2)

SunOS 5.8 Reference Manual

mkfifo(3C) , attributes(5) , stat(3HEAD)

System Interface Guide

NOTES

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

NAME

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

SYNOPSIS

```
#include <unistd.h>
#include <sys/types.h>
int chown(const char * path, uid_t owner, gid_t group);

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

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

DESCRIPTION

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

The `lchown( )` function sets the owner ID and group ID of the named file in the same manner as `chown( )`, unless the named file is a symbolic link. In this case, `lchown( )` changes the ownership of the symbolic link file itself, while `chown( )` changes the ownership of the file or directory to which the symbolic link refers.

If `chown( )`, `lchown( )`, or `fchown( )` is invoked, the set-user- ID and set-group- ID bits of the file mode, `chmod`(2). respectively, are cleared. See `chmod`(2). 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 not in effect, the effective user ID of the process must match the owner of the file or the process must be the super-user to change the ownership of a file. When `_POSIX_CHOWN_RESTRICTED` is in effect, the `chown( )`, `lchown( )`, and `fchown( )` functions 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.

set rstchown = 1

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

set rstchown = 0

See `system`(4) and `fpathconf`(2).

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

RETURN VALUES

`chown()` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

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

`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`	The *path* argument points to an illegal address.
`EINTR`	A signal was caught during the execution of the `chown()` or `lchown()` function.
`EINVAL`	The *group* or *owner* argument is out of range.
`EIO`	An I/O error occurred while reading from or writing to the file system.
`ELOOP`	Too many symbolic links were encountered in translating *path* .
`ENAMETOOLONG`	The length of the *path* argument exceeds `PATH_MAX` , or the length of a *path* component exceeds `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.
`ENOLINK`	The *path* argument points to a remote machine and the link to that machine is no longer active.
`ENOENT`	Either a component of the path prefix or the file referred to by *path* does not exist or is a null pathname.
`ENOTDIR`	A component of the path prefix of *path* is not a directory.

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.

The fchown( ) function will fail if:

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

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

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

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

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

EPERM — The effective user ID does not match the owner of the file, or the process is not the super-user and _POSIX_CHOWN_RESTRICTED indicates that such privilege is required.

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	chown( ) is Async-Signal-Safe

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.

SEE ALSO

Trusted Solaris 8 Reference Manual

`chgrp`(1) , `chown`(1) , `chmod`(2)

SunOS 5.8 Reference Manual

`attributes`(5)

NAME

chroot, fchroot – Change root directory

SYNOPSIS

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

int fchroot(int fildes);
```

DESCRIPTION

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

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

The calling process must assert the `PRIV_PROC_CHROOT` privilege to use this system call. While it is always possible to change to the system root using the `fchroot()` function, it is not guaranteed to succeed in any other case, even should *fildes* be valid in all respects.

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

RETURN VALUES

`chroot()` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `chroot()` function will fail if:

`EACCES` Search permission is denied for a component of the path prefix of *dirname* , or search permission is denied for the directory referred to by *dirname* .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` The *path* argument points to an illegal address.

`EINVAL` The `fchroot()` function 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 execution of the `chroot()` function.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating *path* .
ENAMETOOLONG	The length of the *path* argument exceeds `PATH_MAX` , or the length of a *path* component exceeds `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.
ENOENT	The named directory does not exist or is a null pathname.
ENOLINK	The *path* argument points to a remote machine and the link to that machine is no longer active.
ENOTDIR	Any component of the path name is not a directory.
EPERM	The 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.

SEE ALSO
Trusted Solaris 8 Reference Manual

`chroot`(1M)

WARNINGS

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

NAME

fcntl – File control

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fildes, int cmd, /* arg */ ...);
```

DESCRIPTION

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

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

The available values for *cmd* are defined in the header `<fcntl.h>`, which include:

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

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

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

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

`F_GETFL` Get the file status flags and file access modes, defined in `<fcntl.h>`, for the file description associated with *fildes*. The file access modes can be extracted from the return value using the mask `O_ACCMODE`, which is defined in `<fcntl.h>`. File status flags and file access modes are associated with the

file description and do not affect other file descriptors that refer to the same file with different open file descriptions.

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

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

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

`F_FREESP` Free storage space associated with a section of the ordinary file *fildes*. The section is specified by a variable of data type `struct flock` pointed to by *arg*. The data type `struct flock` is defined in the `<fcntl.h>` header (see `fcntl`(5)) and is described below. Note that all file systems might not support all possible variations of `F_FREESP` arguments. In particular, many file systems allow space to be freed only at the end of a file.

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

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

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

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

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

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

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

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

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

The `flock` structure contains at least the following elements:

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

The value of *l_whence* is SEEK_SET, SEEK_CUR, or SEEK_END, to indicate that the relative offset *l_start* bytes will be measured from the start of the file, current

position or end of the file, respectively. The value of *l_len* is the number of consecutive bytes to be locked. The value of *l_len* may be negative (where the definition of `off_t` permits negative values of *l_len*). After a successful `F_GETLK` or `F_GETLK64` request, that is, one in which a lock was found, the value of *l_whence* will be `SEEK_SET`.

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

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

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

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

All locks associated with a file for a given process are removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process created using `fork`(2)

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

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

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

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

`F_UNSHARE` Remove an existing share reservation.

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

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

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

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

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

Valid *f_access* values are:

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

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

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

Valid *f_deny* values are:

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

F_RDDNY	Set a file share reservation to deny read access to other processes.
F_WRDNY	Set a file share reservation to deny write access to other processes.
F_RWDNY	Set a file share reservation to deny read and write access to other processes.
F_NODNY	Do not deny read or write access to any other process.

RETURN VALUES

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

F_DUPFD	A new file descriptor.
F_GETFD	Value of flags defined in <fcntl.h>. The return value will not be negative.
F_SETFD	Value other than –1.
F_GETFL	Value of file status flags and access modes. The return value will not be negative.
F_SETFL	Value other than –1.
F_GETOWN	Value of the socket owner process or process group; this will not be –1.
F_SETOWN	Value other than –1.
F_FREESP	Value of 0.
F_GETLK	Value other than –1.
F_GETLK64	Value other than –1.
F_SETLK	Value other than –1.
F_SETLK64	Value other than –1.
F_SETLKW	Value other than –1.
F_SETLKW64	Value other than –1.
F_SHARE	Value other than –1.
F_UNSHARE	Value other than –1.

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

ERRORS

The fcntl( ) function will fail if:

EAGAIN	The *cmd* argument is F_SETLK or F_SETLK64, the type of lock (l_type) is a shared (F_RDLCK) or exclusive (F_WRLCK) lock, and the segment of a file to be locked is

already exclusive-locked by another process; or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.

The *cmd* argument is F_FREESP, the file exists, mandatory file / record locking is set, and there are outstanding record locks on the file; or the *cmd* argument is F_SETLK, F_SETLK64, F_SETLKW, or F_SETLKW64, mandatory file / record locking is set, and the file is currently being mapped to virtual memory using mmap(2).

The *cmd* argument is F_SHARE and *f_access* conflicts with an existing *f_deny* share reservation.

EBADF — The *fildes* argument is not a valid open file descriptor; or the *cmd* argument is F_SETLK, F_SETLK64, F_SETLKW, or F_SETLKW64, the type of lock, *l_type*, is a shared lock (F_RDLCK), and *fildes* is not a valid file descriptor open for reading; or the type of lock *l_type* is an exclusive lock (F_WRLCK) and *fildes* is not a valid file descriptor open for writing.

The *cmd* argument is F_FREESP and *fildes* is not a valid file descriptor open for writing.

The *cmd* argument is F_DUP2FD, and *arg* is negative or is not less than the current resource limit for RLIMIT_NOFILE.

The *cmd* argument is F_SHARE, the *f_access* share reservation is for write access, and *fildes* is not a valid file descriptor open for writing.

The *cmd* argument is F_SHARE, the *f_access* share reservation is for read access, and *fildes* is not a valid file descriptor open for reading.

EFAULT — The *cmd* argument is F_GETLK, F_GETLK64, F_SETLK, F_SETLK64, F_SETLKW, F_SETLKW64, or F_FREESP and the *arg* argument points to an illegal address.

The *cmd* argument is F_SHARE or F_UNSHARE and *arg* points to an illegal address.

EINTR — The *cmd* argument is F_SETLKW or F_SETLKW64 and the function was interrupted by a signal.

EINVAL — The *cmd* argument is invalid; or the *cmd* argument is F_DUPFD and *arg* is negative or greater than or equal to

OPEN_MAX; or the *cmd* argument is F_GETLK, F_GETLK64, F_SETLK, F_SETLK64, F_SETLKW, or F_SETLKW64 and the data pointed to by *arg* is not valid; or *fildes* refers to a file that does not support locking.

The *cmd* argument is F_UNSHARE and a reservation with this *f_id* for this process does not exist.

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

EMFILE — The *cmd* argument is F_DUPFD and either OPEN_MAX file descriptors are currently open in the calling process, or no file descriptors greater than or equal to *arg* are available.

ENOLCK — The *cmd* argument is F_SETLK, F_SETLK64, F_SETLKW, or F_SETLKW64 and satisfying the lock or unlock request would result in the number of locked regions in the system exceeding a system-imposed limit.

ENOLINK — Either the *fildes* argument is on a remote machine and the link to that machine is no longer active; or the *cmd* argument is F_FREESP, the file is on a remote machine, and the link to that machine is no longer active.

EOVERFLOW — One of the values to be returned cannot be represented correctly.

The *cmd* argument is F_GETLK, F_SETLK, or F_SETLKW and the smallest or, if *l_len* is non-zero, the largest, offset of any byte in the requested segment cannot be represented correctly in an object of type off_t.

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

The fcntl( ) function may fail if:

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

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

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal Safe

SUMMARY OF TRUSTED SOLARIS CHANGES

For the F_GETLK operation, when the requested lock is blocked, mandatory access checks are required to ensure that the sensitivity label of the calling process that is requesting the lock dominates the sensitivity label of the process holding the blocking lock. This is done to prevent the transmission of lock information from a process holding the blocking lock which dominates the sensitivity label of the calling process making the F_GETLK request. If the calling process fails this MAC check, then fixed results are returned indicating that the entire file is locked, and with zeroes for the process ID and system ID. The calling process may assert the PRIV_FILE_LOCK privilege to bypass this check.

SEE ALSO

Trusted Solaris 8 Reference Manual

lockd(1M), chmod(2), creat(2), exec(2), fork(2), open(2), read(2), write(2)

SunOS 5.8 Reference Manual

close(2), dup(2), pipe(2), fcntl(5)

NOTES

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

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee exclusive access. Files can be accessed without advisory locks, but inconsistencies may result. The network share locking protocol does not support the f_deny value of F_COMPAT. For network file systems, if *f_access* is F_RDACC, *f_deny* is mapped to F_RDDNY. Otherwise, it is mapped to F_RWDNY.

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

If the file server crashes and has to be rebooted, the lock manager (see lockd(1M)) attempts to recover all locks that were associated with that server. If

a lock cannot be reclaimed, the process that held the lock is issued a `SIGLOST` signal.

`read`(2) and `write`(2) system calls on files are affected by mandatory file and record locks. [See `chmod`(2).]

NAME

getcmwfsrange, fgetcmwfsrange – Get file system sensitivity label range

SYNOPSIS

cc [*flags*…] *file*… –ltsol

```
#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* .

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(3C)) 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.

SEE ALSO

Trusted Solaris 8 Reference Manual

pathconf(2)

SunOS 5.8 Reference Manual

sysconf(3C)

NAME

getcmwlabel, lgetcmwlabel, fgetcmwlabel – Get file CMW label

SYNOPSIS

cc [*flags*…] *file* … `–ltsol` [*library*…]

```
#include <tsol/label.h>
int getcmwlabel(char * path, bclabel_t * label_p);

int lgetcmwlabel(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`(2) 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.

label_p is a pointer to an opaque CMW label structure.

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`(2)).

`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 process' 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.

NOTES

Information labels (IL s) are not supported in Trusted Solaris 7 and later releases. Trusted Solaris software interprets any IL s on communications and files from systems running earlier releases as ADMIN_LOW .

Objects still have CMW labels, and CMW labels still include the IL component: IL[SL] ; however, the IL component is fixed at ADMIN_LOW .

As a result, Trusted Solaris 7 has the following characteristics:

- IL s do not display in window labels; SL s (Sensitivity Labels) display alone within brackets.
- IL s do not float.
- Setting an IL on an object has no effect.
- Getting an object's IL will always return ADMIN_LOW .
- Although certain utilities, library functions, and system calls can manipulate IL strings, the resulting IL s are always ADMIN_LOW , and cannot be set on any objects.

SEE ALSO
Trusted Solaris 8 Reference Manual

pathconf(2) , open(2) , setcmwlabel(2)

NAME

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

SYNOPSIS

cc [*flags*…] *file*… `–ltsol`

```
#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* .

RETURN VALUES

These functions return:

`0` On success.

`–1` On failure, and set `errno` 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* .

`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 filesystem 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 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

Trusted Solaris 8 Reference Manual

`setfattrflag`(1), `getfattrflag`(1)

Trusted Solaris Developer's Guide

NAME

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

SYNOPSIS

cc [*flags*…] *file*… –ltsol

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 `PRIV_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 `PRIV_FILE_SETPRIV` must be effective. In addition, only the owner of a file may change its privilege sets, unless the privilege `PRIV_FILE_OWNER` is effective.

The invoking process must have MAC write permission for the named file (unless the privilege `PRIV_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.

Normally MAC read permission is required or the privilege PRIV_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 PRIV_FILE_MAC_READ is not effective then getfpriv( ) returns success but sets the privilege fields of *priv_set* to zero.

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 PRIV_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 PRIV_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 (that is, a file system other than NFS , TMPFS) or (2) an illegal or undefined value is supplied for *op* . Also if privilege PRIV_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) PRIV_FILE_SETPRIV is not effective, or (3) the effective uid does not match the owner of the named file and privilege PRIV_FILE_OWNER is not effective.
EROFS	The named file resides on a read-only file system.

SEE ALSO

Trusted Solaris 8 Reference Manual

getppriv(2) , setppriv(2) , priv_macros(5)

SunOS 5.8 Reference Manual

attributes(5)

NAME

getfsattr, fgetfsattr – Get filesystem security attributes

SYNOPSIS

cc [*flags*…] *file*… `–ltsol`

```
#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_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.

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 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(3C)) while _POSIX_NO_TRUNC is in effect (see pathconf(2)).

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

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

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

getmldadorn, fgetmldadorn – Get file system multilevel directory adornment

SYNOPSIS

cc [*flags*…] *file*… –ltsol

```
#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`(3C)) while `_POSIX_NO_TRUNC` is in effect (see `pathconf`(2)).

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* does not support MLD s and no `mld_prefix` attribute was specified at mount time, no error is returned, and a zero-length string is returned in the *adorn_buf* buffer.

SEE ALSO
Trusted Solaris 8 Reference Manual

`fgetsldname`(2) , `getsldname`(2)

NAME

getsldname, fgetsldname – Get file system single-level directory name

SYNOPSIS

cc [*flags*…] *file*… `–ltsol`

```
#include <tsol/label.h>
int getsldname(char * path_name, bslabel_t * slabel_p, char * name_buf, const int length);

int fgetsldname(int fd, const bslabel_t * slabel_p, char * name_buf, const int length);
```

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.

ATTRIBUTES

See for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtsu

RETURN VALUES

`getsldname( )` and `fgetsldname( )` return:

`0` On success.

`–1` On failure 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(3C)] while _POSIX_NO_TRUNC is in effect. [See pathconf(2) .]

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 MLD s, no error is returned and the first SLD_NAME_MAX bytes in the *name_buf* are cleared.

SEE ALSO

Trusted Solaris 8 Reference Manual

fgetmldadorn(2) , getmldadorn(2)

SunOS 5.8 Reference Manual

sysconf(3C)

NAME

fork, fork1 – Create a new process

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);

pid_t fork1(void);
```

DESCRIPTION

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

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

Scheduling priority and any per-process scheduling parameters that are specific to a given scheduling class may or may not be inherited according to the policy

of that particular class (see priocntl(2)). The child process differs from the parent process in the following ways:

- The child process has a unique process ID which does not match any active process group ID .
- The child process has a different parent process ID (that is, the process ID of the parent process).
- The child process has its own copy of the parent's file descriptors and directory streams. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.
- Each shared memory segment remains attached and the value of shm_nattach is incremented by 1.
- All semadj values are cleared (see semop(2)).
- Process locks, text locks, data locks, and other memory locks are not inherited by the child (see plock(3C) and memcntl(2)).
- The child process's tms structure is cleared: tms_utime , stime , cutime , and cstime are set to 0 (see times(2)).
- The child processes resource utilizations are set to 0; see getrlimit(2) . The it_value and it_interval values for the ITIMER_REAL timer are reset to 0; see getitimer(2) .
- The set of signals pending for the child process is initialized to the empty set.
- Timers created by timer_create(3RT) are not inherited by the child process.
- No asynchronous input or asynchronous output operations are inherited by the child.

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

Solaris Threads

In applications that use the Solaris threads API rather than the POSIX threads API (applications linked with –lthread but not –lpthread), fork( ) duplicates in the child process all threads (see thr_create(3THR)) and LWPs in the parent process. The fork1( ) function duplicates only the calling thread (LWP) in the child process.

POSIX Threads

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

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

fork( ) safety

If a Solaris threads application calls `fork1( )` or a POSIX threads application calls `fork( )`, and the child does more than simply call `exec( )`, there is a possibility of deadlock occurring in the child. The application should use `pthread_atfork`(3THR) to ensure safety with respect to this deadlock. A Solaris threads application must explicitly link with `–lpthread` to access `pthread_atfork( )`. Should there be any outstanding mutexes throughout the process, the application should call `pthread_atfork( )` to wait for and acquire those mutexes prior to calling `fork( )` or `fork1( )`. See "MT-Level of Libraries" on the `attributes`(5) manual page.

RETURN VALUES

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

ERRORS

The `fork( )` function will fail if:

`EAGAIN` The system-imposed limit on the total number of processes under execution by a single user has been exceeded, and the calling process does not have the `PRIV_SYS_MAXPROC` effective privilege, or the total amount of system memory available is temporarily insufficient to duplicate this process.

`ENOMEM` There is not enough swap space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	`fork( )` is Async-Signal-Safe

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

Trusted Solaris 8 Reference Manual

`exec`(2), `fcntl`(2), `getrlimit`(2), `nice`(2), `priocntl`(2), `semop`(2), `shmop`(2), `plock`(3C)

SunOS 5.8 Reference Manual

`alarm`(2), `exit`(2), `getitimer`(2), `memcntl`(2), `mmap`(2), `ptrace`(2), `times`(2), `umask`(2), `wait`(2), `exit`(3C), `pthread_atfork`(3THR), `signal`(3C), `system`(3C), `thr_create`(3THR), `timer_create`(3RT), `attributes`(5), `standards`(5)

NOTES

An applications should call `_exit( )` rather than `exit`(3C) if it cannot `execve( )`, since `exit( )` will flush and close standard I/O channels and

thereby corrupt the parent process's standard I/O data structures. Using `exit`(3C) will flush buffered data twice. See `exit`(2) .

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

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

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

NAME

fork, fork1 – Create a new process

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);

pid_t fork1(void);
```

DESCRIPTION

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

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

Scheduling priority and any per-process scheduling parameters that are specific to a given scheduling class may or may not be inherited according to the policy

of that particular class (see priocntl(2)). The child process differs from the parent process in the following ways:

- The child process has a unique process ID which does not match any active process group ID .
- The child process has a different parent process ID (that is, the process ID of the parent process).
- The child process has its own copy of the parent's file descriptors and directory streams. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.
- Each shared memory segment remains attached and the value of shm_nattach is incremented by 1.
- All semadj values are cleared (see semop(2)).
- Process locks, text locks, data locks, and other memory locks are not inherited by the child (see plock(3C) and memcntl(2)).
- The child process's tms structure is cleared: tms_utime , stime , cutime , and cstime are set to 0 (see times(2)).
- The child processes resource utilizations are set to 0; see getrlimit(2) . The it_value and it_interval values for the ITIMER_REAL timer are reset to 0; see getitimer(2) .
- The set of signals pending for the child process is initialized to the empty set.
- Timers created by timer_create(3RT) are not inherited by the child process.
- No asynchronous input or asynchronous output operations are inherited by the child.

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

Solaris Threads

In applications that use the Solaris threads API rather than the POSIX threads API (applications linked with –lthread but not –lpthread), fork( ) duplicates in the child process all threads (see thr_create(3THR)) and LWPs in the parent process. The fork1( ) function duplicates only the calling thread (LWP) in the child process.

POSIX Threads

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

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

fork( ) safety

If a Solaris threads application calls `fork1( )` or a POSIX threads application calls `fork( )`, and the child does more than simply call `exec( )`, there is a possibility of deadlock occurring in the child. The application should use `pthread_atfork`(3THR) to ensure safety with respect to this deadlock. A Solaris threads application must explicitly link with `–lpthread` to access `pthread_atfork( )`. Should there be any outstanding mutexes throughout the process, the application should call `pthread_atfork( )` to wait for and acquire those mutexes prior to calling `fork( )` or `fork1( )`. See "MT-Level of Libraries" on the `attributes`(5) manual page.

RETURN VALUES

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

ERRORS

The `fork( )` function will fail if:

`EAGAIN` The system-imposed limit on the total number of processes under execution by a single user has been exceeded, and the calling process does not have the `PRIV_SYS_MAXPROC` effective privilege, or the total amount of system memory available is temporarily insufficient to duplicate this process.

`ENOMEM` There is not enough swap space.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	`fork( )` is Async-Signal-Safe

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

Trusted Solaris 8 Reference Manual

`exec`(2), `fcntl`(2), `getrlimit`(2), `nice`(2), `priocntl`(2), `semop`(2), `shmop`(2), `plock`(3C)

SunOS 5.8 Reference Manual

`alarm`(2), `exit`(2), `getitimer`(2), `memcntl`(2), `mmap`(2), `ptrace`(2), `times`(2), `umask`(2), `wait`(2), `exit`(3C), `pthread_atfork`(3THR), `signal`(3C), `system`(3C), `thr_create`(3THR), `timer_create`(3RT), `attributes`(5), `standards`(5)

NOTES

An applications should call `_exit( )` rather than `exit`(3C) if it cannot `execve( )`, since `exit( )` will flush and close standard I/O channels and

thereby corrupt the parent process's standard I/O data structures. Using `exit`(3C) will flush buffered data twice. See `exit`(2) .

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

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

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

NAME

fpathconf, pathconf – Get configurable pathname variables

SYNOPSIS

```
#include <unistd.h>
long int fpathconf(int fildes, int name);

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

DESCRIPTION

The `fpathconf( )` and `pathconf( )` functions provide a method for the application to determine the current value of a configurable limit or option I (variable) that is associated with a file or directory.

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

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

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

Variable	Value of ***name***	Notes
FILESIZEBITS	_PC_FILESIZEBITS	3,4
LINK_MAX	_PC_LINK_MAX	1
MAX_CANON	_PC_MAX_CANON	2
MAX_INPUT	_PC_MAX_INPUT	2
NAME_MAX	_PC_NAME_MAX	3,4
PATH_MAX	_PC_PATH_MAX	4,5
PIPE_BUF	_PC_PIPE_BUF	6
_POSIX_CHOWN_RESTRICTED	_PC_CHOWN_RESTRICTED	7
_POSIX_NO_TRUNC	_PC_NO_TRUNC	3,4
_POSIX_VDISABLE	_PC_VDISABLE	2
_POSIX_ASYNC_IO	_PC_ASYNC_IO	8
_POSIX_PRIO_IO	_PC_PRIO_IO	8
_POSIX_SYNC_IO	_PC_SYNC_IO	8

Notes:

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

2. If *path* or *fildes* does not refer to a terminal file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
3. If *path* or *fildes* refers to a directory, the value returned applies to filenames within the directory.
4. If *path* or *fildes* does not refer to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.
5. If *path* or *fildes* refers to a directory, the value returned is the maximum length of a relative pathname when the specified directory is the working directory.
6. If *path* refers to a FIFO , or *fildes* refers to a pipe or FIFO , the value returned applies to the referenced object. If *path* or *fildes* refers to a directory, the value returned applies to any FIFO that exists or can be created within the directory. If *path* or *fildes* refers to any other type of file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
7. If *path* or *fildes* refers to a directory, the value returned applies to any files, other than directories, that exist or can be created within the directory.
8. If *path* or *fildes* refers to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.

RETURN VALUES

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

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

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

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

ERRORS

The `pathconf( )` function will fail if:

EINVAL	The value of *name* is not valid.
ELOOP	Too many symbolic links were encountered in resolving *path* .

The pathconf( ) function may fail if:

EACCES	Search permission is denied for a component of the path prefix.
EINVAL	The implementation does not support an association of the variable *name* with the specified file.
ENAMETOOLONG	The length of the *path* argument exceeds PATH_MAX or a pathname component is longer than NAME_MAX .
ENAMETOOLONG	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX .
ENOENT	A component of *path* does not name an existing file or *path* is an empty string.
ENOTDIR	A component of the path prefix is not a directory.

The fpathconf( ) function will fail if:

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

The fpathconf( ) function may fail if:

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

EINVAL	The implementation does not support an association of the variable *name* with the specified file.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	pathconf( ) is Async-Signal-Safe

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

SunOS 5.8 Reference Manual

sysconf(3C) , limits(4) , attributes(5) , standards(5)

NAME

setcmwlabel, fsetcmwlabel, lsetcmwlabel – Set CMW label of a file

SYNOPSIS

cc [*flags*…] *file* … `-ltsol` [*library*…]

```
#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 file system that supports the setting of labels on individual objects.

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 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 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* 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 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 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 On success.

–1 On failure, 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 file system that supports the setting of labels on individual objects. 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(3C)] while _POSIX_NO_TRUNC is in effect. See pathconf(2) .
ENOENT	The file referred to by *path* does not exist.
ENOTDIR	A component of the path prefix of *path* is not a directory.
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. A calling process that is not the owner of the file attempted to downgrade the 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 any of these conditions prevails:

`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 file system that supports the setting of labels on individual objects.
	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.
	The calling process is not the owner of the file, attempted to downgrade the 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.
`EPERM`	The calling process does not have mandatory write access to the object referred to by *fd* because the sensitivity label of the object referred to by *fd* 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.
	A calling process that is not the owner of the file attempted to downgrade the sensitivity label associated with the object referred to by *fd* 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 object referred to by *fd* 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 object referred to by *fd* 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

Trusted Solaris 8 Reference Manual

getcmwfsrange(2), getcmwlabel(2)

NAME

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

SYNOPSIS

cc [*flags*…] *file*… `–ltsol`

```
#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* .

RETURN VALUES

These functions return:

`0` On success.

`–1` On failure, and set `errno` 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* .

`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 filesystem 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 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

Trusted Solaris 8 Reference Manual

`setfattrflag`(1), `getfattrflag`(1)

Trusted Solaris Developer's Guide

NAME

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

SYNOPSIS

cc [*flags*…] *file*… –ltsol

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 `PRIV_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 `PRIV_FILE_SETPRIV` must be effective. In addition, only the owner of a file may change its privilege sets, unless the privilege `PRIV_FILE_OWNER` is effective.

The invoking process must have MAC write permission for the named file (unless the privilege `PRIV_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.

Normally MAC read permission is required or the privilege PRIV_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 PRIV_FILE_MAC_READ is not effective then getfpriv( ) returns success but sets the privilege fields of *priv_set* to zero.

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 PRIV_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 PRIV_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 (that is, a file system other than NFS , TMPFS) or (2) an illegal or undefined value is supplied for *op* . Also if privilege PRIV_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) PRIV_FILE_SETPRIV is not effective, or (3) the effective uid does not match the owner of the named file and privilege PRIV_FILE_OWNER is not effective.
EROFS	The named file resides on a read-only file system.

SEE ALSO

Trusted Solaris 8 Reference Manual

getppriv(2) , setppriv(2) , priv_macros(5)

SunOS 5.8 Reference Manual

attributes(5)

NAME

stat, lstat, fstat – Get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char * path, struct stat * buf);

int lstat(const char * path, struct stat * buf);

int fstat(int fildes, struct stat * buf);
```

DESCRIPTION

The `stat()` function obtains information about the file pointed to by *path* . Read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

The `lstat()` function obtains file attributes similar to `stat()` , except when the named file is a symbolic link; in that case `lstat()` returns information about the link, while `stat()` returns information about the file the link references.

The `fstat()` function obtains information about an open file known by the file descriptor *fildes* , obtained from a successful `open`(2) , `creat`(2) , `dup`(2) , `fcntl`(2) , or `pipe`(2) function.

The *buf* argument is a pointer to a `stat` structure into which information is placed concerning the file. A `stat` structure includes the following members:

```
mode_t   st_mode;     /* File mode (see mknod(2)) */
ino_t    st_ino;      /* Inode number */
dev_t    st_dev;      /* ID of device containing */
                      /* a directory entry for this file */
dev_t    st_rdev;     /* ID of device */
                      /* This entry is defined only for */
                      /* char special or block special files */
nlink_t  st_nlink;    /* Number of links */
uid_t    st_uid;      /* User ID of the file's owner */
gid_t    st_gid;      /* Group ID of the file's group */
off_t    st_size;     /* File size in bytes */
time_t   st_atime;    /* Time of last access */
time_t   st_mtime;    /* Time of last data modification */
time_t   st_ctime;    /* Time of last file status change */
                      /* Times measured in seconds since */
                      /* 00:00:00 UTC, Jan. 1, 1970 */
long     st_blksize;  /* Preferred I/O block size */
blkcnt_t st_blocks;   /* Number of 512 byte blocks allocated*/
```

Descriptions of structure members are as follows:

`st_mode` The mode of the file as described in `mknod`(2) . In addition to the modes described in `mknod()` , the mode of a file may also be `S_IFLNK` if the file is a symbolic link. `S_IFLNK` may only be returned by `lstat()` .

st_ino	This field uniquely identifies the file in a given file system. The pair st_ino and st_dev uniquely identifies regular files.
st_dev	This field uniquely identifies the file system that contains the file. Its value may be used as input to the ustat( ) function to determine more information about this file system. No other meaning is associated with this value.
st_rdev	This field should be used only by administrative commands. It is valid only for block special or character special files and only has meaning on the system where the file was configured.
st_nlink	This field should be used only by administrative commands.
st_uid	The user ID of the file's owner.
st_gid	The group ID of the file's group.
st_size	For regular files, this is the address of the end of the file. For block special or character special, this is not defined. See also pipe(2) .
st_atime	Time when file data was last accessed. Changed by the following functions: creat( ) , mknod( ) , pipe( ) , utime(2) , and read(2) .
st_mtime	Time when data was last modified. Changed by the following functions: creat( ) , mknod( ) , pipe( ) , utime( ) , and write(2) .
st_ctime	Time when file status was last changed. Changed by the following functions: chmod( ) , chown( ) , creat( ) , link(2) , mknod( ) , pipe( ) , unlink(2) , 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.

stat( ) , lstat( ) , and fstat( ) require mandatory read access to the final component of *path* . If the file descriptor is open only for writing, fstat( ) requires mandatory read access to the object to which the descriptor refers. To override these restrictions, the calling process may assert the PRIV_FILE_MAC_READ privilege in its set of effective privileges.

If the calling process does not have mandatory read access, `stat()`, `lstat()`, and `fstat()` return fixed values for some elements of the `stat` structure.

RETURN VALUES

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

ERRORS

The `stat()`, `fstat()`, and `lstat()` functions will fail if:

`EOVERFLOW` The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by *buf* .

The `stat()` and `lstat()` functions will fail if:

`EACCES` Search permission is denied for a component of the path prefix. 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` The *buf* or *path* argument points to an illegal address.

`EINTR` A signal was caught during the execution of the `stat()` or `lstat()` function.

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

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

`ENOENT` The named file does not exist or is the null pathname.

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

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

`EOVERFLOW` A component is too large to store in the structure pointed to by *buf* .

The `fstat()` function will fail if:

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

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

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

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

EOVERFLOW A component is too large to store in the structure pointed to by *buf* .

USAGE

The stat( ) , fstat( ) , and lstat( ) functions have transitional interfaces for 64-bit file offsets. See lf64(5) .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	stat( ) and fstat( ) are Async-Signal-Safe

SUMMARY OF TRUSTED SOLARIS CHANGES

stat( ) , lstat( ) , and fstat( ) require mandatory read access to the final component of *path* . If the file descriptor is open only for writing, fstat( ) requires mandatory read access to the object to which the descriptor refers. To override these restrictions, the calling process may assert the PRIV_FILE_MAC_READ privilege in its set of effective privileges.

To override access restrictions, the calling process of stat( ) or lstat( ) may also assert one or both of these privileges: PRIV_FILE_DAC_SEARCH and PRIV_FILE_MAC_SEARCH .

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

Trusted Solaris 8 Reference Manual

chmod(2) , chown(2) , creat(2) , fcntl(2) , link(2) , mknod(2) , open(2) , read(2) , unlink(2) , write(2)

SunOS 5.8 Reference Manual

dup(2) , pipe(2) , time(2) , utime(2) , fattach(3C) , stat(3HEAD) , attributes(5)

NOTES

If you use chmod(2) to change the file group owner permissions on a file with ACL entries, both the file group owner permissions and the ACL mask are changed to the new permissions. Be aware that the new ACL mask permissions may change the effective permissions for additional users and groups who have ACL entries on the file.

NAME

statvfs, fstatvfs – Get file system information

SYNOPSIS

```
#include <sys/types.h>
#include <sys/statvfs.h>
int statvfs(const char * path, struct statvfs * buf);

int fstatvfs(int fildes, struct statvfs * buf);
```

DESCRIPTION

The `statvfs( )` function returns a "generic superblock" describing a file system; it can be used to acquire information about mounted file systems. The *buf* argument is a pointer to a structure (described below) that is filled by the function.

The *path* argument should name a file that resides on that file system. The file system type is known to the operating system. Read, write, or execute permission for the named file is not required, but all directories listed in the path name leading to the file must be searchable.

The `statvfs` structure pointed to by *buf* includes the following members:

```
u_long      f_bsize;             /* preferred file system block size */
u_long      f_frsize;            /* fundamental filesystem block
                                    (size if supported) */
fsblkcnt_t  f_blocks;            /* total # of blocks on file system
                                    in units of f_frsize */
fsblkcnt_t  f_bfree;             /* total # of free blocks */
fsblkcnt_t  f_bavail;            /* # of free blocks avail to
                                    non-super-user */
fsfilcnt_t  f_files;             /* total # of file nodes (inodes) */
fsfilcnt_t  f_ffree;             /* total # of free file nodes */
fsfilcnt_t  f_favail;            /* # of inodes avail to
                                    non-super-user*/
u_long      f_fsid;              /* file system id (dev for now) */
char        f_basetype[FSTYPSZ]; /* target fs type name,
                                    null-terminated */
u_long      f_flag;              /* bit mask of flags */
u_long      f_namemax;           /* maximum file name length */
char        f_fstr[32];          /* file system specific string */
u_long      f_filler[16];        /* reserved for future expansion */
```

The `f_basetype` member contains a null-terminated FSType name of the mounted target.

The following values can be returned in the `f_flag` field:

```
ST_RDONLY    0x01    /* read-only file system */
ST_NOSUID    0x02    /* does not support setuid/setgid semantics */
ST_NOTRUNC   0x04    /* does not truncate file names longer than
                        NAME_MAX */
```

The `fstatvfs( )` function is similar to `statvfs( )` , except that the file named by *path* in `statvfs( )` is instead identified by an open file descriptor *fildes* obtained from a successful `open`(2) , `creat`(2) , `dup`(2) , `fcntl`(2) , or `pipe`(2) function call.

RETURN VALUES

`statvfs( )` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `statvfs( )` and `fstatvfs( )` functions will fail if:

`EOVERFLOW` One of the values to be returned cannot be represented correctly in the structure pointed to by *buf* .

The `statvfs( )` function will fail if:

`EACCES` Search permission is denied on a component of the path prefix. 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` The *path* or *buf* argument points to an illegal address.

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

`EIO` An I/O error occurred while reading the file system.

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

`ENAMETOOLONG` The length of a *path* component exceeds `NAME_MAX` characters, or the length of *path* The exceeds `PATH_MAX` characters.

`ENOENT` Either a component of the path prefix or the file referred to by *path* does not exist.

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

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

The `fstatvfs()` function will fail if:

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	The *fildes* argument is not an open file descriptor.
EFAULT	The *buf* argument points to an illegal address.
EINTR	A signal was caught during the execution of the `fstatvfs()` function.
EIO	An I/O error occurred while reading the file system.

USAGE

The `statvfs()` and `fstatvfs()` functions have transitional interfaces for 64-bit file offsets. See `lf64`(5) .

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

Trusted Solaris 8 Reference Manual

`chmod`(2) , `chown`(2) , `creat`(2) , `fcntl`(2) , `link`(2) , `mknod`(2) , `open`(2) , `read`(2) , `unlink`(2) , `write`(2)

SunOS 5.8 Reference Manual

`dup`(2) , `pipe`(2) , `time`(2) , `utime`(2)

BUGS

The values returned for `f_files` , `f_ffree` , and `f_favail` may not be valid for NFS mounted file systems.

NAME

getaudit, setaudit, getaudit_addr, setaudit_addr – 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*);

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

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

DESCRIPTION

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

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

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

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

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

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

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

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

`getaudit( )` and `setaudit( )` return:

`0` On success.

`–1` On failure, and set `errno` to indicate the error.

ERRORS

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

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

`EPERM` The process did not have the appropriate privilege.

USAGE

Only processes with the appropriate privileges may successfully execute these calls.

SUMMARY OF TRUSTED SOLARIS CHANGES

This functionality is active only if auditing is enabled. By default, auditing is enabled in the Trusted Solaris environment. See *Trusted Solaris Audit Administration* for more information.

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

SEE ALSO

Trusted Solaris 8 Reference Manual

`audit`(2)

NAME

getaudit, setaudit, getaudit_addr, setaudit_addr – 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*);

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

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

DESCRIPTION

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

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

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

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

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

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

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

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

`getaudit( )` and `setaudit( )` return:

`0` On success.

`–1` On failure, and set `errno` to indicate the error.

ERRORS

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

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

`EPERM` The process did not have the appropriate privilege.

USAGE

Only processes with the appropriate privileges may successfully execute these calls.

SUMMARY OF TRUSTED SOLARIS CHANGES

This functionality is active only if auditing is enabled. By default, auditing is enabled in the Trusted Solaris environment. See *Trusted Solaris Audit Administration* for more information.

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

SEE ALSO

Trusted Solaris 8 Reference Manual

`audit`(2)

NAME

getauid, setauid – Get and set user audit identity

SYNOPSIS

cc [*flags*…] *file* … –lbsm –lsocket –lnsl –lintl [*library*…]

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

#include <bsm/audit.h>
int getauid(au_id_t * auid);

int setauid(au_id_t * auid);
```

DESCRIPTION

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

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

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

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

ERRORS

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

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

`EPERM` The process does not have the appropriate privileges.

SUMMARY OF TRUSTED SOLARIS CHANGES

This functionality is active only if auditing is enabled. By default, auditing is enabled in the Trusted Solaris environment. See *Trusted Solaris Audit Administration* for more information.

The privileges explained in `DESCRIPTION` are needed to run this command successfully.

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

SEE ALSO

Trusted Solaris 8 Reference Manual

`audit`(2) , `getaudit`(2)

NAME

getclearance – Get process clearance

SYNOPSIS

cc [*flags*…] *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

`getclearance()` returns:

`0` On success.

`–1` On failure, 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

Trusted Solaris 8 Reference Manual

`setclearance`(2)

NAME

getcmwfsrange, fgetcmwfsrange – Get file system sensitivity label range

SYNOPSIS

cc [*flags*…] *file*… –ltsol

```
#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* .

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(3C)) 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.

SEE ALSO

Trusted Solaris 8 Reference Manual

pathconf(2)

SunOS 5.8 Reference Manual

sysconf(3C)

NAME

getcmwlabel, lgetcmwlabel, fgetcmwlabel – Get file CMW label

SYNOPSIS

cc [*flags*…] *file* … `–ltsol` [*library*…]

```
#include <tsol/label.h>
int getcmwlabel(char * path, bclabel_t * label_p);

int lgetcmwlabel(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`(2) 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.

label_p is a pointer to an opaque CMW label structure.

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`(2)).

`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 process' 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.

NOTES

Information labels (IL s) are not supported in Trusted Solaris 7 and later releases. Trusted Solaris software interprets any IL s on communications and files from systems running earlier releases as ADMIN_LOW .

Objects still have CMW labels, and CMW labels still include the IL component: IL[SL] ; however, the IL component is fixed at ADMIN_LOW .

As a result, Trusted Solaris 7 has the following characteristics:

- IL s do not display in window labels; SL s (Sensitivity Labels) display alone within brackets.
- IL s do not float.
- Setting an IL on an object has no effect.
- Getting an object's IL will always return ADMIN_LOW .
- Although certain utilities, library functions, and system calls can manipulate IL strings, the resulting IL s are always ADMIN_LOW , and cannot be set on any objects.

SEE ALSO
Trusted Solaris 8 Reference Manual

pathconf(2) , open(2) , setcmwlabel(2)

NAME

getcmwplabel – Get process CMW label

SYNOPSIS

cc [*flags*…] *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

`getcmwplabel( )` returns:

`0` On success.

`–1` On failure, 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`(2)

NAME

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

SYNOPSIS

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

DESCRIPTION

The `getdents( )` function attempts to read *nbyte* bytes from the directory associated with the file descriptor *fildes* and to format them as file system independent directory entries in the buffer pointed to by *buf*. Since the file system independent directory entries are of variable lengths, in most cases the actual number of bytes returned will be less than *nbyte*. The file system independent directory entry is specified by the `dirent` structure. See `dirent`(3HEAD).

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

RETURN VALUES

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

ERRORS

The `getdents( )` function will fail if:

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

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

`EINVAL` The *nbyte* argument is not large enough for one directory entry.

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

`ENOENT` The current file pointer for the directory is not located at a valid entry.

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

`ENOTDIR` The *fildes* argument is not a directory.

`EOVERFLOW`	The value of the `dirent` structure member `d_ino` or `d_off` cannot be represented in an `ino_t` or `off_t`.

USAGE

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

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

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

SunOS 5.8 Reference Manual

`readdir`(3C), `dirent`(3HEAD), `lf64`(5)

NAME

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

SYNOPSIS

cc [*flags…*] *file…* –ltsol

#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* .

RETURN VALUES

These functions return:

`0` On success.

`–1` On failure, and set `errno` 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* .

`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 filesystem 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 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

Trusted Solaris 8 Reference Manual

`setfattrflag`(1) , `getfattrflag`(1)

Trusted Solaris Developer's Guide

NAME

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

SYNOPSIS

cc [*flags*…] *file*… –ltsol

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 `PRIV_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 `PRIV_FILE_SETPRIV` must be effective. In addition, only the owner of a file may change its privilege sets, unless the privilege `PRIV_FILE_OWNER` is effective.

The invoking process must have MAC write permission for the named file (unless the privilege `PRIV_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.

Normally MAC read permission is required or the privilege PRIV_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 PRIV_FILE_MAC_READ is not effective then getfpriv( ) returns success but sets the privilege fields of *priv_set* to zero.

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 PRIV_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 PRIV_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 (that is, a file system other than NFS , TMPFS) or (2) an illegal or undefined value is supplied for *op* . Also if privilege PRIV_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) PRIV_FILE_SETPRIV is not effective, or (3) the effective uid does not match the owner of the named file and privilege PRIV_FILE_OWNER is not effective.
EROFS	The named file resides on a read-only file system.

SEE ALSO

Trusted Solaris 8 Reference Manual

getppriv(2) , setppriv(2) , priv_macros(5)

SunOS 5.8 Reference Manual

attributes(5)

NAME

getfsattr, fgetfsattr – Get filesystem security attributes

SYNOPSIS

cc [*flags*…] *file*… –ltsol

#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_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.

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 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(3C)) while _POSIX_NO_TRUNC is in effect (see pathconf(2)).

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

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

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

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

The `setgroups()` function sets the supplementary group access list of the calling process from the array of group ID s specified by *grouplist* . The number of entries is specified by *ngroups* and can not 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 ID s set for the calling process and `setgroups()` returns `0` . Otherwise, `–1` is returned and `errno` is set to indicate the error.

ERRORS

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

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

The `getgroups()` function will fail if:

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

The `setgroups()` function will fail if:

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

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

ATTRIBUTES

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

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

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

Trusted Solaris 8 Reference Manual

`chown`(2) , `setuid`(2)

SunOS 5.8 Reference Manual

`groups`(1) , `getuid`(2) , `getgrnam`(3C) , `initgroups`(3C) , `attributes`(5)

NAME

getmldadorn, fgetmldadorn – Get file system multilevel directory adornment

SYNOPSIS

cc [*flags*…] *file*… –ltsol

```
#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`(3C)) while `_POSIX_NO_TRUNC` is in effect (see `pathconf`(2)).

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* does not support MLD s and no `mld_prefix` attribute was specified at mount time, no error is returned, and a zero-length string is returned in the *adorn_buf* buffer.

SEE ALSO
Trusted Solaris 8 Reference Manual

`fgetsldname`(2) , `getsldname`(2)

NAME

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

SYNOPSIS

cc [*flags*…] *file* … `–ltsol` [*library*…]

```
#include <sys/tsol/ipcl.h>
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.

`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.

RETURN VALUES

`getmsgqcmwlabel( )` , `getshmcmwlabel( )` , and `getsemcmwlabel( )` return:

`0` On success.

`–1` On failure, and sets `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
Trusted Solaris 8 Reference Manual

`msgget`(2) , `semget`(2) , `shmget`(2)

NAME

getpattr, setpattr – Get / set process attribute flags

SYNOPSIS

cc [*flags*…] *file*… –ltsol

#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 , PAF_TP_OFF) 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.

setpattr(PAF_TRUSTED_PATH, PAF_TP_OFF)

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

`getpattr()` and `setpattr()` return:

`0` On success.

`–1` On failure, 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

Trusted Solaris 8 Reference Manual

`pattr`(1)

NAME

getpid, getpgrp, getppid, getpgid – Get process, process group, and parent process IDs

SYNOPSIS

```
#include <unistd.h>
pid_t getpid(void);

pid_t getpgrp(void);

pid_t getppid(void);

pid_t getpgid(pid_t pid);
```

DESCRIPTION

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

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

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

The `getpgid()` function returns the process group ID of the process whose process ID is equal to *pid* , or the process group ID of the calling process, if *pid* is equal to 0.

RETURN VALUES

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

ERRORS

The `getpgid()` function will fail if:

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

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

The `getpgid()` function may fail if:

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

ATTRIBUTES

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

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

SUMMARY OF TRUSTED SOLARIS CHANGES

MAC and DAC policies are added to the `getpgid()` command. To avoid covert channel issues, the Trusted Solaris environment does not distinguish between failures due to policy and those due to nonexistence of the target process.

SEE ALSO

Trusted Solaris 8 Reference Manual

`exec`(2) , `fork`(2) , `getsid`(2) , `intro`(3)

SunOS 5.8 Reference Manual

`setpgid`(2) , `setpgrp`(2) , `setsid`(2) , `signal`(3C) , `attributes`(5)

NAME

getpid, getpgrp, getppid, getpgid – Get process, process group, and parent process IDs

SYNOPSIS

```
#include <unistd.h>
pid_t getpid(void);

pid_t getpgrp(void);

pid_t getppid(void);

pid_t getpgid(pid_t pid);
```

DESCRIPTION

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

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

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

The `getpgid()` function returns the process group ID of the process whose process ID is equal to *pid* , or the process group ID of the calling process, if *pid* is equal to 0.

RETURN VALUES

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

ERRORS

The `getpgid()` function will fail if:

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

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

The `getpgid()` function may fail if:

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

ATTRIBUTES

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

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

SUMMARY OF TRUSTED SOLARIS CHANGES

MAC and DAC policies are added to the `getpgid()` command. To avoid covert channel issues, the Trusted Solaris environment does not distinguish between failures due to policy and those due to nonexistence of the target process.

SEE ALSO

Trusted Solaris 8 Reference Manual

`exec`(2) , `fork`(2) , `getsid`(2) , `intro`(3)

SunOS 5.8 Reference Manual

`setpgid`(2) , `setpgrp`(2) , `setsid`(2) , `signal`(3C) , `attributes`(5)

NAME

getpid, getpgrp, getppid, getpgid – Get process, process group, and parent process IDs

SYNOPSIS

```
#include <unistd.h>
pid_t getpid(void);

pid_t getpgrp(void);

pid_t getppid(void);

pid_t getpgid(pid_t pid);
```

DESCRIPTION

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

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

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

The `getpgid()` function returns the process group ID of the process whose process ID is equal to *pid* , or the process group ID of the calling process, if *pid* is equal to 0.

RETURN VALUES

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

ERRORS

The `getpgid()` function will fail if:

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

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

The `getpgid()` function may fail if:

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

ATTRIBUTES

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

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

SUMMARY OF TRUSTED SOLARIS CHANGES

MAC and DAC policies are added to the `getpgid()` command. To avoid covert channel issues, the Trusted Solaris environment does not distinguish between failures due to policy and those due to nonexistence of the target process.

SEE ALSO

Trusted Solaris 8 Reference Manual

`exec`(2) , `fork`(2) , `getsid`(2) , `intro`(3)

SunOS 5.8 Reference Manual

`setpgid`(2) , `setpgrp`(2) , `setsid`(2) , `signal`(3C) , `attributes`(5)

NAME

getpid, getpgrp, getppid, getpgid – Get process, process group, and parent process IDs

SYNOPSIS

```
#include <unistd.h>
pid_t getpid(void);

pid_t getpgrp(void);

pid_t getppid(void);

pid_t getpgid(pid_t pid);
```

DESCRIPTION

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

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

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

The `getpgid()` function returns the process group ID of the process whose process ID is equal to *pid* , or the process group ID of the calling process, if *pid* is equal to 0.

RETURN VALUES

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

ERRORS

The `getpgid()` function will fail if:

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

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

The `getpgid()` function may fail if:

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

ATTRIBUTES

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

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

SUMMARY OF TRUSTED SOLARIS CHANGES

MAC and DAC policies are added to the `getpgid()` command. To avoid covert channel issues, the Trusted Solaris environment does not distinguish between failures due to policy and those due to nonexistence of the target process.

SEE ALSO

Trusted Solaris 8 Reference Manual

`exec`(2) , `fork`(2) , `getsid`(2) , `intro`(3)

SunOS 5.8 Reference Manual

`setpgid`(2) , `setpgrp`(2) , `setsid`(2) , `signal`(3C) , `attributes`(5)

NAME

getppriv, setppriv – Return or assign a privilege set associated with the invoking process

SYNOPSIS

cc [*flags*…] *file*… –ltsol

```
#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` On success.

`–1` On failure, 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

Trusted Solaris 8 Reference Manual

`getfpriv`(2) , `setfpriv`(2) , `priv_to_str`(3TSOL) , `priv_set_to_str`(3TSOL) , `str_to_priv`(3TSOL) , `str_to_priv_set`(3TSOL) , `priv_macros`(5)

NAME

getrlimit, setrlimit – Control maximum system resource consumption

SYNOPSIS

```
#include <sys/resource.h>
int getrlimit(int resource, struct rlimit * rlp);

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

DESCRIPTION

Limits on the consumption of a variety of system resources by a process and each process it creates may be obtained with the `getrlimit( )` and set with `setrlimit( )` functions.

Each call to either `getrlimit( )` or `setrlimit( )` identifies a specific resource to be operated upon as well as a resource limit. A resource limit is a pair of values: one specifying the current (soft) limit, the other a maximum (hard) limit. Soft limits may be changed by a process to any value that is less than or equal to the hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or equal to the soft limit. Only a process 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` . The *rlp* argument is a pointer to `struct rlimit` that includes the following members:

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

The type `rlim_t` is an arithmetic data type to which objects of type `int` , `size_t` , and `off_t` can be cast without loss of information.

The possible resources, their descriptions, and the actions taken when the current limit is exceeded are summarized as follows:

`RLIMIT_CORE` The maximum size of a core file in bytes that may be created by a process. A limit of `0` will prevent the creation of a core file. The writing of a core file will terminate at this size.

`RLIMIT_CPU` The maximum amount of CPU time in seconds used by a process. This is a soft limit only. The `SIGXCPU` signal is sent to the process. If the process is holding or ignoring `SIGXCPU` , the behavior is scheduling class defined.

`RLIMIT_DATA` The maximum size of a process's heap in bytes. The `brk`(2) function will fail with `errno` set to `ENOMEM` .

`RLIMIT_FSIZE` The maximum size of a file in bytes that may be created by a process. A limit of `0` will prevent the creation of a file. The `SIGXFSZ` signal is sent to the process. If the process is holding or ignoring `SIGXFSZ` , continued attempts to

increase the size of a file beyond the limit will fail with `errno` set to `EFBIG` .

`RLIMIT_NOFILE` One more than the maximum value that the system may assign to a newly created descriptor. This limit constrains the number of file descriptors that a process may create.

`RLIMIT_STACK` The maximum size of a process's stack in bytes. The system will not automatically grow the stack beyond this limit.

Within a process, `setrlimit( )` will increase the limit on the size of your stack, but will not move current memory segments to allow for that growth. To guarantee that the process stack can grow to the limit, the limit must be altered prior to the execution of the process in which the new stack size is to be used.

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

The `SIGSEGV` signal is sent to the process. If the process is holding or ignoring `SIGSEGV` , or is catching `SIGSEGV` and has not made arrangements to use an alternate stack (see `sigaltstack`(2)), the disposition of `SIGSEGV` will be set to `SIG_DFL` before it is sent.

`RLIMIT_VMEM` The maximum size of a process's mapped address space in bytes. If this limit is exceeded, the `brk`(2) and `mmap`(2) functions will fail with `errno` set to `ENOMEM` . In addition, the automatic stack growth will fail with the effects outlined above.

`RLIMIT_AS` This is the maximum size of a process's total available memory, in bytes. If this limit is exceeded, the `brk`(2) , `malloc`(3C) , `mmap`(2) and `sbrk`(2) functions will fail with `errno` set to `ENOMEM` . In addition, the automatic stack growth will fail with the effects outlined above.

Because limit information is stored in the per-process information, the shell builtin `ulimit` command must directly execute this system call if it is to affect all future processes created by the shell.

The value of the current limit of the following resources affect these implementation defined parameters:

Limit	Implementation Defined Constant
RLIMIT_FSIZE	FCHR_MAX
RLIMIT_NOFILE	OPEN_MAX

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

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

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

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

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

RETURN VALUES

`getrlimit()` and `setrlimit()` return:

`0` On success.

`–1` On failure, and set `errno` to indicate the error.

ERRORS

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

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

`EINVAL` An invalid *resource* was specified; or in a `setrlimit()` call, the new `rlim_cur` exceeds the new `rlim_max`.

`EPERM` The limit specified to `setrlimit()` would have raised the maximum limit value, and the calling process does not have the `PRIV_SYS_CONFIG` privilege.

The `setrlimit()` function may fail if:

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

USAGE

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

SUMMARY OF TRUSTED SOLARIS CHANGES

The calling process must have the `PRIV_SYS_CONFIG` privilege in order to increase a hard resource limit.

SEE ALSO

Trusted Solaris 8 Reference Manual

`open`(2)

SunOS 5.8 Reference Manual

`brk`(2) , `sigaltstack`(2) , `malloc`(3C) , `signal`(3C) , `signal`(5)

NAME

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

SYNOPSIS

cc [*flags*…] *file* … –ltsol [*library*…]

#include <sys/tsol/ipcl.h>
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.

`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.

RETURN VALUES

`getmsgqcmwlabel( )` , `getshmcmwlabel( )` , and `getsemcmwlabel( )` return:

`0` On success.

`–1` On failure, and sets `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

Trusted Solaris 8 Reference Manual

`msgget`(2) , `semget`(2) , `shmget`(2)

NAME

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

SYNOPSIS

cc [*flags*…] *file* … –ltsol [*library*…]

#include <sys/tsol/ipcl.h>
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.

`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.

RETURN VALUES

`getmsgqcmwlabel( )` , `getshmcmwlabel( )` , and `getsemcmwlabel( )` return:

`0` On success.

`–1` On failure, and sets `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

Trusted Solaris 8 Reference Manual

`msgget`(2) , `semget`(2) , `shmget`(2)

NAME

getsid – Get process group ID of session leader

SYNOPSIS

```
#include <unistd.h>
pid_t getsid(pid_t pid);
```

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( )` returns the process group ID of the session leader of the specified process. Otherwise, it returns `(pid_t)–1` and sets `errno` to indicate the error.

ERRORS

The `getsid( )` function will fail if:

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

`ESRCH` There is no process with a process ID equal to *pid*. 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 `getsid( )` system call.

SEE ALSO

Trusted Solaris 8 Reference Manual

`intro`(2), `exec`(2), `fork`(2), `getpid`(2), `setpgid`(2)

NAME

getsldname, fgetsldname – Get file system single-level directory name

SYNOPSIS

cc [*flags*…] *file*… –ltsol

#include <tsol/label.h>
int **getsldname**(char * *path_name*, bslabel_t * *slabel_p*, char * *name_buf*, const int *length*);

int **fgetsldname**(int *fd*, const bslabel_t * *slabel_p*, char * *name_buf*, const int *length*);

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.

ATTRIBUTES

See for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtsu

RETURN VALUES

getsldname( ) and fgetsldname( ) return:

0 On success.

–1 On failure 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(3C)] while _POSIX_NO_TRUNC is in effect. [See pathconf(2) .]

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 MLD s, no error is returned and the first SLD_NAME_MAX bytes in the *name_buf* are cleared.

SEE ALSO

Trusted Solaris 8 Reference Manual

fgetmldadorn(2) , getmldadorn(2)

SunOS 5.8 Reference Manual

sysconf(3C)

NAME

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

SYNOPSIS

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

DESCRIPTION

`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`(3HEAD)), 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.

RETURN VALUES

`kill( )` returns:

`0` On success.

`–1` On failure, does not send a signal, and sets `errno` to indicate the error.

ERRORS

The `kill( )` function will fail if:

`EINVAL` The *sig* argument 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*.

USAGE

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

ATTRIBUTES

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

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

SUMMARY OF TRUSTED SOLARIS CHANGES

Process MAC write policy and the process owner policy is checked.

SEE ALSO

Trusted Solaris 8 Reference Manual

`exec`(2), `getpid`(2), `getsid`(2), `sigsend`(2), `intro`(3)

SunOS 5.8 Reference Manual

`kill`(1), `setpgrp`(2), `sigaction`(2), `signal`(3C), `signal`(3HEAD), `attributes`(5)

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);
```

DESCRIPTION

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

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

If chown( ) , lchown( ) , or fchown( ) is invoked, the set-user- ID and set-group- ID bits of the file mode, chmod(2) . respectively, are cleared. See chmod(2) . 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 not in effect, the effective user ID of the process must match the owner of the file or the process must be the super-user to change the ownership of a file. When _POSIX_CHOWN_RESTRICTED is in effect, the chown( ) , lchown( ) , and fchown( ) functions 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.

set rstchown = 1

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

set rstchown = 0

See system(4) and fpathconf(2) .

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

RETURN VALUES

`chown()` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

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

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

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

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

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

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

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

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

`ENOENT` Either a component of the path prefix or the file referred to by *path* does not exist or is a null pathname.

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

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.

The fchown( ) function will fail if:

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	chown( ) is Async-Signal-Safe

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.

SEE ALSO

Trusted Solaris 8 Reference Manual

`chgrp`(1) , `chown`(1) , `chmod`(2)

SunOS 5.8 Reference Manual

`attributes`(5)

NAME

getcmwlabel, lgetcmwlabel, fgetcmwlabel – Get file CMW label

SYNOPSIS

cc [*flags*…] *file* … –ltsol [*library*…]

```
#include <tsol/label.h>
int getcmwlabel(char * path, bclabel_t * label_p);

int lgetcmwlabel(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`(2) 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.

label_p is a pointer to an opaque CMW label structure.

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`(2)).

`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 process' 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.

NOTES

Information labels (IL s) are not supported in Trusted Solaris 7 and later releases. Trusted Solaris software interprets any IL s on communications and files from systems running earlier releases as ADMIN_LOW .

Objects still have CMW labels, and CMW labels still include the IL component: IL[SL] ; however, the IL component is fixed at ADMIN_LOW .

As a result, Trusted Solaris 7 has the following characteristics:

- IL s do not display in window labels; SL s (Sensitivity Labels) display alone within brackets.
- IL s do not float.
- Setting an IL on an object has no effect.
- Getting an object's IL will always return ADMIN_LOW .
- Although certain utilities, library functions, and system calls can manipulate IL strings, the resulting IL s are always ADMIN_LOW , and cannot be set on any objects.

SEE ALSO
Trusted Solaris 8 Reference Manual

pathconf(2) , open(2) , setcmwlabel(2)

NAME

link – Link to a file

SYNOPSIS

```
#include <unistd.h>
int link(const char *existing, const char *new);
```

DESCRIPTION

The `link()` function creates a new link (directory entry) for the existing file and increments its link count by one. The *existing* argument points to a path name naming an existing file. The *new* argument points to a pathname naming the new directory entry to be created.

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, `0` is returned. Otherwise, `−1` is returned, no link is created, and errno is set to indicate the `error`.

ERRORS

The `link()` function will fail if:

`EACCES` A component of either path prefix denies search permission. 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 where the entry for the new link is being placed cannot be extended because the user's quota of disk blocks on that file system has been exhausted.

`EEXIST` The link named by *new* exists.

`EFAULT` The *existing* or *new* argument points to an illegal address.

EINTR	A signal was caught during the execution of the `link()` function.
ELOOP	Too many symbolic links were encountered in translating *path*.
EMLINK	The maximum number of links to a file would be exceeded.
ENAMETOOLONG	The length of the *existing* or *new* argument exceeds `PATH_MAX`, or the length of a *existing* or *new* component exceeds `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.
ENOENT	The *existing* or *new* argument is a null pathname; a component of either path prefix does not exist; or the file named by *existing* does not exist.
ENOLINK	The *existing* or *new* argument points to a remote machine and the link to that machine is no longer active.
ENOSPC	The directory that would contain the link cannot be extended.
ENOTDIR	A component of either path prefix is not a directory.
EPERM	The file named by *existing* is a directory and the 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).

ATTRIBUTES

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

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

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
Trusted Solaris 8 Reference Manual

`symlink`(2), `unlink`(2)

NAME

llseek – Move extended read / write file pointer

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
offset_t llseek(int fildes, offset_t offset, int whence);
```

DESCRIPTION

The `llseek()` function sets the 64-bit extended file pointer associated with the open file descriptor specified by *fildes* as follows:

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

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` privilege to bypass this check. If mandatory read access is not granted, this system call succeeds, but offset data is not returned.

RETURN VALUES

Upon successful completion, `llseek()` returns the resulting pointer location as measured in bytes from the beginning of the file. Remote file descriptors are the only ones that allow negative file pointers. Otherwise, `–1` is returned, the file pointer remains unchanged, and `errno` is set to indicate the error.

ERRORS

The `llseek()` function will fail if:

`EBADF`	The *fildes* argument is not an open file descriptor.
`EINVAL`	The *whence* argument is not `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`; the *offset* argument is not a valid offset for this file system type; or the *fildes* argument is not a remote file descriptor, and the resulting file pointer would be negative.
`ESPIPE`	The *fildes* argument is associated with a pipe or FIFO.

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.

SEE ALSO

Trusted Solaris 8 Reference Manual

`creat`(2), `fcntl`(2), `lseek`(2), `open`(2)

SunOS 5.8 Reference Manual

`dup`(2)

NAME

lseek – Move read / write file pointer

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

DESCRIPTION

The `lseek()` function sets the file pointer associated with the open file descriptor specified by *fildes* as follows:

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

The symbolic constants `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` are defined in the header `<unistd.h>`.

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

The `lseek()` function allows the file pointer to be set beyond the existing data in the file. If data are later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return bytes of value 0 until data are written into the gap.

If *fildes* is a remote file descriptor and *offset* is negative, `lseek()` returns the file pointer even if it is negative. The `lseek()` function will not, by itself, extend the size of a file.

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.

RETURN VALUES

Upon successful completion, the resulting offset, as measured in bytes from the beginning of the file, is returned. Otherwise, `(off_t)-1` is returned, the file offset remains unchanged, and `errno` is set to indicate the error.

ERRORS

The `lseek()` function will fail if:

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

`EINVAL` The *whence* argument is not `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`; or the *fildes* argument is not a remote file descriptor and the resulting file pointer would be negative.

EOVERFLOW	The resulting file offset would be a value which cannot be represented correctly in an object of type off_t for regular files.
ESPIPE	The *fildes* argument is associated with a pipe, a FIFO, or a socket.

USAGE

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

In multithreaded applications, using lseek( ) in conjunction with a read(2) or write(2) call on a file descriptor shared by more than one thread is not an atomic operation. To ensure atomicity, use pread( ) or pwrite( ).

ATTRIBUTES

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

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

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.

SEE ALSO

Trusted Solaris 8 Reference Manual

creat(2), fcntl(2), lseek(2), open(2)

SunOS 5.8 Reference Manual

dup(2)

NAME

setcmwlabel, fsetcmwlabel, lsetcmwlabel – Set CMW label of a file

SYNOPSIS

cc [*flags*…] *file* … `-ltsol` [*library*…]

```
#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 file system that supports the setting of labels on individual objects.

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 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 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* 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 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 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 On success.

–1 On failure, 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 file system that supports the setting of labels on individual objects. 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(3C)] while _POSIX_NO_TRUNC is in effect. See pathconf(2) .
ENOENT	The file referred to by *path* does not exist.
ENOTDIR	A component of the path prefix of *path* is not a directory.
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. A calling process that is not the owner of the file attempted to downgrade the 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 any of these conditions prevails:

`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 file system that supports the setting of labels on individual objects.
	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.
	The calling process is not the owner of the file, attempted to downgrade the 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.
`EPERM`	The calling process does not have mandatory write access to the object referred to by *fd* because the sensitivity label of the object referred to by *fd* 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.
	A calling process that is not the owner of the file attempted to downgrade the sensitivity label associated with the object referred to by *fd* 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 object referred to by *fd* 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 object referred to by *fd* 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
Trusted Solaris 8 Reference Manual

`getcmwfsrange`(2) , `getcmwlabel`(2)

NAME

stat, lstat, fstat – Get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char * path, struct stat * buf);

int lstat(const char * path, struct stat * buf);

int fstat(int fildes, struct stat * buf);
```

DESCRIPTION

The `stat()` function obtains information about the file pointed to by *path* . Read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

The `lstat()` function obtains file attributes similar to `stat()` , except when the named file is a symbolic link; in that case `lstat()` returns information about the link, while `stat()` returns information about the file the link references.

The `fstat()` function obtains information about an open file known by the file descriptor *fildes* , obtained from a successful `open`(2) , `creat`(2) , `dup`(2) , `fcntl`(2) , or `pipe`(2) function.

The *buf* argument is a pointer to a `stat` structure into which information is placed concerning the file. A `stat` structure includes the following members:

```
mode_t   st_mode;     /* File mode (see mknod(2)) */
ino_t    st_ino;      /* Inode number */
dev_t    st_dev;      /* ID of device containing */
                      /* a directory entry for this file */
dev_t    st_rdev;     /* ID of device */
                      /* This entry is defined only for */
                      /* char special or block special files */
nlink_t  st_nlink;    /* Number of links */
uid_t    st_uid;      /* User ID of the file's owner */
gid_t    st_gid;      /* Group ID of the file's group */
off_t    st_size;     /* File size in bytes */
time_t   st_atime;    /* Time of last access */
time_t   st_mtime;    /* Time of last data modification */
time_t   st_ctime;    /* Time of last file status change */
                      /* Times measured in seconds since */
                      /* 00:00:00 UTC, Jan. 1, 1970 */
long     st_blksize;  /* Preferred I/O block size */
blkcnt_t st_blocks;   /* Number of 512 byte blocks allocated*/
```

Descriptions of structure members are as follows:

`st_mode` The mode of the file as described in `mknod`(2) . In addition to the modes described in `mknod()` , the mode of a file may also be `S_IFLNK` if the file is a symbolic link. `S_IFLNK` may only be returned by `lstat()` .

st_ino	This field uniquely identifies the file in a given file system. The pair st_ino and st_dev uniquely identifies regular files.
st_dev	This field uniquely identifies the file system that contains the file. Its value may be used as input to the ustat( ) function to determine more information about this file system. No other meaning is associated with this value.
st_rdev	This field should be used only by administrative commands. It is valid only for block special or character special files and only has meaning on the system where the file was configured.
st_nlink	This field should be used only by administrative commands.
st_uid	The user ID of the file's owner.
st_gid	The group ID of the file's group.
st_size	For regular files, this is the address of the end of the file. For block special or character special, this is not defined. See also pipe(2) .
st_atime	Time when file data was last accessed. Changed by the following functions: creat( ) , mknod( ) , pipe( ) , utime(2) , and read(2) .
st_mtime	Time when data was last modified. Changed by the following functions: creat( ) , mknod( ) , pipe( ) , utime( ) , and write(2) .
st_ctime	Time when file status was last changed. Changed by the following functions: chmod( ) , chown( ) , creat( ) , link(2) , mknod( ) , pipe( ) , unlink(2) , 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.

stat( ) , lstat( ) , and fstat( ) require mandatory read access to the final component of *path* . If the file descriptor is open only for writing, fstat( ) requires mandatory read access to the object to which the descriptor refers. To override these restrictions, the calling process may assert the PRIV_FILE_MAC_READ privilege in its set of effective privileges.

If the calling process does not have mandatory read access, `stat()`, `lstat()`, and `fstat()` return fixed values for some elements of the `stat` structure.

RETURN VALUES

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

ERRORS

The `stat()`, `fstat()`, and `lstat()` functions will fail if:

`EOVERFLOW` The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by *buf*.

The `stat()` and `lstat()` functions will fail if:

`EACCES` Search permission is denied for a component of the path prefix. 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` The *buf* or *path* argument points to an illegal address.

`EINTR` A signal was caught during the execution of the `stat()` or `lstat()` function.

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

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

`ENOENT` The named file does not exist or is the null pathname.

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

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

`EOVERFLOW` A component is too large to store in the structure pointed to by *buf*.

The `fstat()` function will fail if:

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

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

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

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

EOVERFLOW A component is too large to store in the structure pointed to by *buf* .

USAGE

The stat() , fstat() , and lstat() functions have transitional interfaces for 64-bit file offsets. See lf64(5) .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	stat() and fstat() are Async-Signal-Safe

SUMMARY OF TRUSTED SOLARIS CHANGES

stat() , lstat() , and fstat() require mandatory read access to the final component of *path* . If the file descriptor is open only for writing, fstat() requires mandatory read access to the object to which the descriptor refers. To override these restrictions, the calling process may assert the PRIV_FILE_MAC_READ privilege in its set of effective privileges.

To override access restrictions, the calling process of stat() or lstat() may also assert one or both of these privileges: PRIV_FILE_DAC_SEARCH and PRIV_FILE_MAC_SEARCH .

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

Trusted Solaris 8 Reference Manual

chmod(2) , chown(2) , creat(2) , fcntl(2) , link(2) , mknod(2) , open(2) , read(2) , unlink(2) , write(2)

SunOS 5.8 Reference Manual

dup(2) , pipe(2) , time(2) , utime(2) , fattach(3C) , stat(3HEAD) , attributes(5)

NOTES

If you use chmod(2) to change the file group owner permissions on a file with ACL entries, both the file group owner permissions and the ACL mask are changed to the new permissions. Be aware that the new ACL mask permissions may change the effective permissions for additional users and groups who have ACL entries on the file.

NAME

mkdir – Make a directory

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(const char *path, mode_t mode);
```

DESCRIPTION

The `mkdir()` function creates a new directory named by the path name pointed to by *path*. The mode of the new directory is initialized from *mode* (see `chmod`(2) for values of mode). The protection part of the *mode* argument is modified by the process' file creation mask (see `umask`(2)).

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to the process's effective group ID, or if the `S_ISGID` bit is set in the parent directory, then the group ID of the directory is inherited from the parent. The `S_ISGID` bit of the new directory is inherited from the parent directory.

If *path* is a symbolic link, it is not followed.

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

Upon successful completion, `mkdir()` marks for update the `st_atime`, `st_ctime`, and `st_mtime` fields of the directory. Also, the `st_ctime` and `st_mtime` fields of the directory that contains the new entry are marked for update. 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`(2) system call.

Trusted Solaris distinguishes directories with sensitivity labels from unlabeled directories through special prefixes called adornments that are appended to the beginning of the directory's name. See `setfsattr`(1M). A multilevel directory has the default adornment "`.MLD.`"; a single-level directory has the adornment "`.SLD.n/`", where n is a number. If the directory name includes the multilevel adornment, the directory will be created as a multilevel directory, provided all other conditions for success are met. 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.

RETURN VALUES

`mkdir()` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `mkdir()` function will fail if:

`EACCES`	Either a component of the path prefix denies search permission or write permission is denied on the parent directory of the directory to be created. 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 where the new file entry is being placed cannot be extended because the user's quota of disk blocks on that file system has been exhausted; the new directory cannot be created because the user's quota of disk blocks on that file system has been exhausted; or the user's quota of inodes on the file system where the file is being created has been exhausted.
`EEXIST`	The named file already exists.
`EFAULT`	The *path* argument points to an illegal address.
`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.
`ENAMETOOLONG`	The length of the *path* argument exceeds `PATH_MAX`, or the length of a *path* component

	exceeds NAME_MAX while _POSIX_NO_TRUNC is in effect.
ENOENT	A component of the path prefix does not exist or is a null pathname.
ENOLINK	The *path* argument points to a remote machine and the link to that machine is no longer active.
ENOSPC	No free space is available on the device containing the directory.
ENOTDIR	A component of the path prefix is not a directory.
EROFS	The path prefix resides on a read-only file system.

ATTRIBUTES

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

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

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks as described under File Access in Intro(2). Trusted Solaris distinguishes directories with sensitivity labels from unlabeled directories through special prefixes called adornments.

SEE ALSO

Trusted Solaris 8 Reference Manual

adornfc(1M), chmod(2), mknod(2)

SunOS 5.8 Reference Manual

umask(2), stat(5)

NAME

mknod – Make a directory, or a special or ordinary file

SYNOPSIS

#include <sys/stat.h>
int **mknod**(const char **path*, mode_t *mode*, dev_t *dev*);

DESCRIPTION

The `mknod( )` function creates a new file named by the path name pointed to by *path*. The file type and permissions of the new file are initialized from *mode*. This system call will not create an object in a multilevel directory. Single-level directories are automatically created during path-name lookup and `getsldname`(2).

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 file type is specified in *mode* by the `S_IFMT` bits, which must be set to one of the following values:

`S_IFIFO`	fifo special
`S_IFCHR`	character special
`S_IFDIR`	directory
`S_IFBLK`	block special
`S_IFREG`	ordinary file

The file access permissions are specified in *mode* by the 0007777 bits, and may be constructed by a bitwise OR operation of the following values:

`S_ISUID`	04000	Set user ID on execution.
`S_ISGID`	020#0	Set group ID on execution if # is `7`, `5`, `3`, or `1`. Enable mandatory file / record locking if # is `6`, `4`, `2`, or `0`
`S_ISVTX`	01000	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.
S_ISVTX		On directories, restricted deletion flag.

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 process' file mode creation mask: all bits set in the process' file mode creation mask are cleared (see umask(2)). If *mode* indicates a block or character special file, *dev* is a configuration-dependent specification of a character or block I/O device. If *mode* does not indicate a block special or character special device, *dev* is ignored. See makedev(3C).

If *path* is a symbolic link, it is not followed.

RETURN VALUES

mknod( ) returns:

0 On success.

–1 On failure, and sets errno to indicate the error.

ERRORS

The mknod( ) function will fail if:

EACCES 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 where the new file entry is being placed cannot be extended because the user's quota of disk blocks on that file system has been exhausted, or the user's quota of inodes on the file system where the file is being created has been exhausted.
EEXIST	The named file exists.
EFAULT	The *path* argument points to an illegal address.
EINTR	A signal was caught during the execution of the `mknod()` function.
EINVAL	An invalid argument exists.
EIO	An I/O error occurred while accessing the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of the *path* argument exceeds `PATH_MAX`, or the length of a *path* component exceeds `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.
ENOENT	A component of the path prefix specified by *path* does not name an existing directory or *path* is an empty string.
ENOLINK	The *path* argument points to a remote machine and the link to that machine is no longer active.
ENOSPC	The directory that would contain the new file cannot be extended or the file system is out of file allocation resources.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	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.

The `mknod()` function may fail if:

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

USAGE

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

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

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

Trusted Solaris 8 Reference Manual

chmod(2), exec(2), mkdir(2)

SunOS 5.8 Reference Manual

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

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

SYNOPSIS

cc [*flags…*] *file…* –ltsol

#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* .

RETURN VALUES

These functions return:

`0` On success.

`–1` On failure, and set `errno` 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* .
`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 filesystem 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 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

Trusted Solaris 8 Reference Manual

`setfattrflag`(1) , `getfattrflag`(1)

Trusted Solaris Developer's Guide

NAME

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

SYNOPSIS

cc [*flags*…] *file*… `–ltsol`

```
#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* .

RETURN VALUES

These functions return:

`0` On success.

`–1` On failure, and set `errno` 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* .

`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 filesystem 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 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

Trusted Solaris 8 Reference Manual

`setfattrflag`(1), `getfattrflag`(1)

Trusted Solaris Developer's Guide

NAME

mount – Mount a file system

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mount.h>
#include <sys/mntent.h>
int mount(const char *spec, const char *dir, int mflag, char *fstype, char *dataptr, int datalen,
char *optptr, int optlen);
```

DESCRIPTION

The `mount()` function requests that a removable file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. The *spec* and *dir* arguments are pointers to path names. After a successful call to `mount()`, all references to the file *dir* refer to the root directory on the mounted file system. The mounted file system is inserted into the kernel list of all mounted file systems. This list can be examined through the mounted file system table (see `mnttab`(4)). The *fstype* argument is the file system type name. Standard file system names are defined with the prefix `MNTTYPE_` in `<sys/mntent.h>`. The *dataptr* argument is 0 if no file system-specific data is to be passed; otherwise it points to an area of size *datalen* that contains the file system-specific data for this mount and the `MS_DATA` flag should be set. If the `MS_OPTIONSTR` flag is set, then *optptr* points to a buffer containing the list of options to be used for this mount. The *optlen* argument specifies the length of the buffer. On completion of the `mount()` call, the options in effect for the mounted file system are returned in this buffer. If `MS_OPTIONSTR` is not specified, then the options for this mount will not appear in the mounted file systems table. The *mflag* argument is constructed by a bitwise-inclusive-OR of flags from the following list, defined in `<sys/mount.h>`.

`MS_DATA` — If this flag is set, the *dataptr* and *datalen* arguments describe a block of file system-specific binary data at address *dataptr* of length *datalen*. This is interpreted by file system-specific code within the operating system and its format depends on the file system type. If a particular file system type does not require this data, *dataptr* and *datalen* should both be 0.

`MS_OPTIONSTR` — If this flag is set, the *optptr* and *optlen* arguments describe a character buffer at address *optptr* of size *optlen*. When calling `mount()`, the character buffer should contain a null-terminated string of options to be passed to the file system-specific code within the operating system. On a successful return, the file system-specific code will return the list of options recognized. Unrecognized options are ignored. The format of the string is a list of option names separated by commas. Options that

	have values (rather than binary options such as `suid` or `nosuid`), are separated by "=" such as `dev=2c4046c`. Standard option names are defined in `<sys/mntent.h>`. The `slabel`, `low_range`, and `hi_range` values must be hexadecimal strings starting with `0x` followed by exactly 68 hex digits. The allowed and forced values must be hexadecimal strings starting with `0x` followed by exactly 32 hex digits. Only strings defined in the "C" locale are supported. The maximum length option string that can be passed to or returned from a `mount( )` call is defined by the `MAX_MNTOPT_STR` constant. The buffer should be long enough to contain more options than were passed in, as the state of any default options that were not passed in the input option string may also be returned in the recognized options list that is returned.
`MS_RDONLY`	The file system should be mounted for reading only. This flag should also be specified for file systems that are incapable of writing (for example, CDROM). Without this flag, writing is permitted according to individual file accessibility.
`MS_NOSUID`	This option prevents programs that are marked set-user-ID or set-group-ID from executing (see `chmod`(1)). It also causes `open`(2) to return `ENXIO` when attempting to open block or character special files.
`MS_REMOUNT`	Remounts a read-only file system as read-write.
`MS_OVERLAY`	Allow the file system to be mounted over an existing file system mounted on *dir*, making the underlying file system inaccessible. If a mount is attempted on a pre-existing mount point without setting this flag, the mount will fail.
`MS_GLOBAL`	Mount a file system globally if the system is configured and booted as part of a cluster (see `clinfo`(1M)).

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. When mounting a UFS file system, the calling process should assert the PRIV_SYS_FS_CONFIG privilege. Otherwise, the mount succeeds, but logging is not enabled / disabled, errno is set to EPERM, and the user sees an error message.

RETURN VALUES

mount( ) returns:

0 On success.

–1 On failure, and sets errno to indicate the error.

ERRORS

The mount( ) function will fail if:

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

EFAULT The *spec*, *dir*, *fstype*, or *dataptr* argument points outside the allocated address space of the process.

EINVAL The super block has an invalid magic number or the *fstype* is invalid.

ELOOP Too many symbolic links were encountered in translating *spec* or *dir*.

ENAMETOOLONG The length of the *path* argument exceeds PATH_MAX, or the length of a *path* component exceeds NAME_MAX while _POSIX_NO_TRUNC is in effect.

ENOENT None of the named files exists or is a null pathname.

ENOLINK	The *path* argument points to a remote machine and the link to that machine is no longer active.
ENOSPC	The file system state in the super block is not FsOKAY and *mflag* requests write permission.
ENOTBLK	The *spec* argument is not a block special device.
ENOTDIR	The *dir* argument is not a directory, or a component of a path prefix is not a directory.
ENOTSUP	A global mount is attempted (the MS_GLOBAL flag is set in *mflag*) on a machine which is not booted as a cluster or a local mount is attempted and *dir* is within a globally mounted file system.
ENXIO	The device associated with *spec* does not exist.
EOVERFLOW	The length of the option string to be returned in the *dataptr* argument exceeds the size of the buffer specified by *datalen*.
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	The *spec* argument is remote and cannot be mounted.
EROFS	The *spec* argument is write protected and *mflag* requests write permission.

The mount( ) function will succeed, but logging will not be enabled / disabled if:

EPERM	*dir* is a UFS file system and the calling process has not asserted the PRIV_SYS_FS_CONFIG privilege.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access, logging, 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. When mounting a UFS file system, the calling

process should assert the PRIV_SYS_FS_CONFIG privilege. Otherwise, the mount succeeds, but logging is not enabled / disabled and errno is set to EPERM.

SEE ALSO

Trusted Solaris 8 Reference Manual

mount(1M), umount(2), mnttab(4)

NOTES

MS_OPTIONSTR-type option strings should be used.

Some flag bits set file system options that can also be passed in an option string. Options are first set from the option string with the last setting of an option in the string determining the value to be set by the option string. Any options controlled by flags are then applied, overriding any value set by the option string.

NAME

msgctl – Message control operations

SYNOPSIS

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

DESCRIPTION

The `msgctl( )` function provides a variety of message control operations as specified by *cmd*. The following *cmd*s are available:

`IPC_STAT` Place the current value of each member of the data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in `intro`(2).

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 the following members of the data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode /* access permission bits only */
msg_qbytes
```

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

`msgctl( )` returns:

`0` On success.

–1 On failure, and sets `errno` to indicate the error.

ERRORS

The `msgctl( )` function will fail if:

`EACCES` *cmd* is `IPC_STAT`, 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.

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

`EINVAL` The *msqid* argument is not a valid message queue identifier; or the *cmd* argument is not a valid command or is `IPC_SET` and `msg_perm.uid` or `msg_perm.gid` is not valid.

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

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

Trusted Solaris 8 Reference Manual

`intro`(2), `msgget`(2)

NAME

msgget, msggetl – Get message queue

SYNOPSIS

```
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
cc [flags...] 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 to the effective user ID and effective group ID , respectively, of the calling process.
- The low-order 9 bits of `msg_perm.mode` are set to the low-order 9 bits of *msgflg* .
- `msg_qnum` , `msg_lspid` , `msg_lrpid` , `msg_stime` , and `msg_rtime` are set to 0.
- `msg_ctime` is set to the current time.
- `msg_qbytes` is set to the system limit.

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

Upon successful completion, a non-negative integer representing a message queue identifier is returned. Otherwise, –1 is returned and `errno` is set to indicate the error.

ERRORS

The `msgget()` function will fail if:

`EACCES` A 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 system wide 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

Trusted Solaris 8 Reference Manual

`intro`(2) , `msgctl`(2)

SunOS 5.8 Reference Manual

`stdio`(3C)

NAME

msgget, msggetl – Get message queue

SYNOPSIS

```
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
cc [flags...] 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 to the effective user ID and effective group ID , respectively, of the calling process.
- The low-order 9 bits of `msg_perm.mode` are set to the low-order 9 bits of *msgflg* .
- `msg_qnum` , `msg_lspid` , `msg_lrpid` , `msg_stime` , and `msg_rtime` are set to 0.
- `msg_ctime` is set to the current time.
- `msg_qbytes` is set to the system limit.

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

Upon successful completion, a non-negative integer representing a message queue identifier is returned. Otherwise, –1 is returned and `errno` is set to indicate the error.

ERRORS

The `msgget()` function will fail if:

`EACCES` A 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 system wide 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

Trusted Solaris 8 Reference Manual

`intro`(2) , `msgctl`(2)

SunOS 5.8 Reference Manual

`stdio`(3C)

NAME

msgrcv – Message receive operation

SYNOPSIS

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

int msgrcv( int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

DESCRIPTION

The `msgrcv()` function reads a message from the queue associated with the message queue identifier specified by *msqid* and places it in the user-defined buffer pointed to by *msgp*.

The *msgp* argument points to a user-defined buffer that must contain first a field of type `long int` that will specify the type of the message, and then a data portion that will hold the data bytes of the message.

The structure below is an example of what this user-defined buffer might look like:

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

The `mtype` member is the received message's type as specified by the sending process.

The `mtext` member is the text of the message.

The *msgsz* argument specifies the size in bytes of `mtext`. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg*&`MSG_NOERROR`) is non-zero. The truncated part of the message is lost and no indication of the truncation is given to the calling process.

The *msgtyp* argument specifies the type of message requested as follows:

- If *msgtyp* is 0, the first message on the queue is received.
- If *msgtyp* is greater than 0, the first message of type *msgtyp* is received.
- If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

The *msgflg* argument specifies which of the following actions is to be taken if a message of the desired type is not on the queue:

- If (*msgflg*&`IPC_NOWAIT`) is non-zero, the calling process will return immediately with a return value of `–1` and `errno` set to `ENOMSG`.
- If (*msgflg*&`IPC_NOWAIT`) is 0, the calling process will suspend execution until one of the following occurs:
  - A message of the desired type is placed on the queue.

- The message queue identifier *msqid* is removed from the system (see `msgctl`(2)); when this occurs, `errno` is set equal to `EIDRM` and `–1` is returned.
- The calling process receives a signal that is to be caught; in this case a message is not received and the calling process resumes execution in the manner prescribed in `sigaction`(2).

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* (see `intro`(2)):

- `msg_qnum` is decremented by 1.
- `msg_lrpid` is set equal to the process ID of the calling process.
- `msg_rtime` is set equal to the current time.

RETURN VALUES

Upon successful completion, `msgrcv( )` returns a value equal to the number of bytes actually placed into the buffer *mtext*. Otherwise, `–1` is returned, no message is received, and `errno` is set to indicate the error.

ERRORS

The `msgrcv( )` function will fail if:

`E2BIG` The value of `mtext` is greater than *msgsz* and (*msgflg*&`MSG_NOERROR`) is 0.

`EACCES` Operation permission is denied to the calling process, and the calling process does not have the appropriate privilege override(s) in its set of effective privileges. See `intro`(2).

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.

`EIDRM` The message queue identifier *msqid* is removed from the system.

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

`EINVAL` The *msqid* argument is not a valid message queue identifier; or the value of *msgsz* is less than 0.

`ENOMSG` The queue does not contain a message of the desired type and (*msgflg*&`IPC_NOWAIT`) is non-zero.

The `msgrcv( )` function may fail if:

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

USAGE

The value passed as the *msgp* argument should be converted to type `void *`.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

Trusted Solaris 8 Reference Manual

`intro`(2), `msgctl`(2), `msgget`(2), `msgsnd`(2)

SunOS 5.8 Reference Manual

`sigaction`(2), `signal`(3C)

NAME

msgsnd – Message send operation

SYNOPSIS

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

int msgsnd( int msqid, const void *msgp, size_t msgsz, int msgflg);
```

DESCRIPTION

The `msgsnd()` function is used to send a message to the queue associated with the message queue identifier specified by *msqid*.

The *msgp* argument points to a user-defined buffer that must contain first a field of type `long int` that will specify the type of the message, and then a data portion that will hold the data bytes of the message.

`msgsnd()` requires either that a process have discretionary and mandatory write access to *msqid*, or that the effective privilege set of the calling process includes `PRIV_IPC_DAC_WRITE` and `PRIV_IPC_MAC_WRITE`.

The structure below is an example of what this user-defined buffer might look like:

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

The `mtype` member is a non-zero positive type `long int` that can be used by the receiving process for message selection.

The `mtext` member is any text of length *msgsz* bytes. The *msgsz* argument can range from 0 to a system-imposed maximum.

The *msgflg* argument specifies the action to be taken if one or more of the following are true:

- The number of bytes already on the queue is equal to `msg_qbytes`; see `intro`(2).
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

- If (*msgflg*`&IPC_NOWAIT`) is non-zero, the message will not be sent and the calling process will return immediately.
- If (*msgflg*`&IPC_NOWAIT`) is 0, the calling process will suspend execution until one of the following occurs:
- The condition responsible for the suspension no longer exists, in which case the message is sent.

- The message queue identifier *msqid* is removed from the system (see `msgctl`(2)); when this occurs, `errno` is set equal to `EIDRM` and `–1` is returned.
- The calling process receives a signal that is to be caught; in this case the message is not sent and the calling process resumes execution in the manner prescribed in `sigaction`(2).

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid* (see `intro`(2)):

- `msg_qnum` is incremented by 1.
- `msg_lspid` is set equal to the process ID of the calling process.
- `msg_stime` is set equal to the current time.

RETURN VALUES

`msgsnd( )` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `msgsnd( )` function will fail if:

`EACCES` Operation permission is denied to the calling process, and the process did not have the appropriate privilege in its set of effective privileges. See `intro`(2).

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 above and (*msgflg*`&IPC_NOWAIT`) is non-zero.

`EIDRM` The message queue identifier *msgid* is removed from the system.

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

`EINVAL` The value of *msqid* is not a valid message queue identifier, or the value of `mtype` is less than 1; or the value of *msgsz* is less than 0 or greater than the system-imposed limit.

The `msgsnd( )` function may fail if:

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

USAGE

The value passed as the *msgp* argument should be converted to type `void *`.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

Trusted Solaris 8 Reference Manual

`intro`(2), `msgctl`(2), `msgget`(2), `msgrcv`(2)

SunOS 5.8 Reference Manual

`sigaction`(2), `signal`(3C)

NAME

nice – Change priority of a process

SYNOPSIS

```
#include <unistd.h>
int nice(int incr);
```

DESCRIPTION

The `nice( )` function allows a process to change its priority. The invoking process must be in a scheduling class that supports the `nice( )`.

The `nice( )` function adds the value of *incr* to the nice value of the calling process. A process's nice value is a non-negative number for which a greater positive value results in lower CPU priority.

A maximum nice value of (2 * `NZERO`)–1 and a minimum nice value of 0 are imposed by the system. `NZERO` is defined in `<limits.h>` with a default value of 20. Requests for values above or below these limits result in the nice value being set to the corresponding limit. A nice value of 40 is treated as 39.

RETURN VALUES

Upon successful completion, `nice( )` returns the new nice value minus `NZERO`. Otherwise, `–1` is returned, the process's *nice* value is not changed, and `errno` is set to indicate the error.

ERRORS

The `nice( )` function will fail if:

`EINVAL` The `nice( )` function is called by a process in a scheduling class other than time-sharing.

`EPERM` *incr* is negative or greater than 40 and the `PRIV_SYS_CONFIG` privilege of the calling process is not asserted.

USAGE

The `priocntl`(2) function is a more general interface to scheduler functions.

Since `–1` is a permissible return value in a successful situation, an application wishing to check for error situations should set `errno` to `0`, then call `nice( )`, and if it returns `–1`, check to see if `errno` is non-zero.

SUMMARY OF TRUSTED SOLARIS CHANGES

Use of the `PRIV_SYS_CONFIG` privilege replaces the check for super-user.

SEE ALSO

Trusted Solaris 8 Reference Manual

`exec`(2), `priocntl`(2)

SunOS 5.8 Reference Manual

`nice`(1)

NAME

open – Open a file

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int oflag, /* mode_t mode */ ...);
```

DESCRIPTION

The `open()` function establishes the connection between a file and a file descriptor. It creates an open file description that refers to a file and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to that file. The *path* argument points to a pathname naming the file.

The `open()` function returns a file descriptor for the named file that is the lowest file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor does not share it with any other process in the system. The `FD_CLOEXEC` file descriptor flag associated with the new file descriptor is cleared.

The file offset used to mark the current position within the file is set to the beginning of the file.

The file status flags and file access modes of the open file description is set according to the value of *oflag*.

Values for *oflag* are constructed by a bitwise-inclusive-OR of flags from the following list, defined in `<fcntl.h>`. Applications must specify exactly one of the first three values (file access modes) below in the value of *oflag*:

`O_RDONLY` Open for reading only.

`O_WRONLY` Open for writing only.

`O_RDWR` Open for reading and writing. The result is undefined if this flag is applied to a FIFO.

Any combination of the following may be used:

`O_APPEND` If set, the file offset is set to the end of the file prior to each write.

`O_CREAT` 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(2).]

- All bits set in the file mode-creation mask of the process are cleared. [See umask(2).]
- The "save text image after execution bit" of the mode is cleared. [See chmod(2) .] O_SYNC write I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion. [See fcntl(5) definition of O_SYNC .]
- The calling process must assert the PRIV_SYS_CONFIG privilege to override clearing the S_ISVTX bit.

O_DSYNC — Write I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion.

O_EXCL — If O_CREAT and O_EXCL are set, open( ) fails if the file exists. The check for the existence of the file and the creation of the file if it does not exist is atomic with respect to other processes executing open( ) naming the same filename in the same directory with O_EXCL and O_CREAT set. If O_CREAT is not set, the effect is undefined.

O_LARGEFILE — If set, the offset maximum in the open file description is the largest value that can be represented correctly in an object of type off64_t.

O_NOCTTY — If set and *path* identifies a terminal device, open( ) does not cause the terminal device to become the controlling terminal for the process.

O_NONBLOCK or O_NDELAY

These flags may affect subsequent reads and writes (see read(2) and write(2)). If both O_NDELAY and O_NONBLOCK are set, O_NONBLOCK takes precedence.

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NONBLOCK or O_NDELAY is set:

An open( ) for reading only returns without delay. An open( ) for writing only returns an error if no process currently has the file open for reading.

If O_NONBLOCK and O_NDELAY are clear:

An open( ) for reading only blocks until a process opens the file for writing. An open( ) for writing only blocks until a process opens the file for reading.

After both ends of a FIFO have been opened, there is no guarantee that further calls to open( ) O_RDONLY (O_WRONLY) will synchronize with later calls to open( ) O_WRONLY (O_RDONLY) until both ends of the FIFO have been closed by all readers and writers. Any data written into a FIFO will be lost if both ends of the FIFO are closed before the data is read.

When opening a block special or character special file that supports non-blocking opens:

If O_NONBLOCK or O_NDELAY is set:

The open( ) function returns without blocking for the device to be ready or available. Subsequent behavior of the device is device-specific.

If O_NONBLOCK and O_NDELAY are clear:

The open( ) function blocks until the device is ready or available before returning.

Otherwise, the behavior of O_NONBLOCK and O_NDELAY is unspecified.

O_RSYNC — Read I/O operations on the file descriptor complete at the same level of integrity as specified by the O_DSYNC and O_SYNC flags. If both O_DSYNC and O_RSYNC are set in *oflag*, all I/O operations on the file descriptor complete as defined by synchronized I/O data integrity completion. If both O_SYNC and O_RSYNC are set in *oflag*, all I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.

O_SYNC — When opening a regular file, this flag affects subsequent writes. If set, each write(2) will wait for both the file data and file status to be physically updated. Write I/O operations on the file descriptor complete as defined by synchronized I/O file integrity completion.

O_TRUNC — If the file exists and is a regular file, and the file is successfully opened O_RDWR or O_WRONLY, its length is truncated to 0 and the mode and owner are unchanged. It has no effect on FIFO special files or terminal device files. Its effect on other file types is implementation-dependent. The result of using O_TRUNC with O_RDONLY is undefined.

If O_CREAT is set and the file did not previously exist, upon successful completion, open( ) marks for update the st_atime, st_ctime, and

st_mtime fields of the file and the st_ctime and st_mtime fields of the parent directory.

If O_TRUNC is set and the file did previously exist, upon successful completion, open( ) marks for update the st_ctime and st_mtime fields of the file.

If *path* refers to a STREAMS file, *oflag* may be constructed from O_NONBLOCK or O_NODELAY OR-ed with either O_RDONLY, O_WRONLY, or O_RDWR. Other flag values are not applicable to STREAMS devices and have no effect on them. The values O_NONBLOCK and O_NODELAY affect the operation of STREAMS drivers and certain functions (see read(2) and write(2)) applied to file descriptors associated with STREAMS files. For STREAMS drivers, the implementation of O_NONBLOCK and O_NODELAY is device-specific.

When open( ) is invoked to open a named stream, and the connld module (see connld(7M)) has been pushed on the pipe, open( ) blocks until the server process has issued an I_RECVFD ioctl( ) (see streamio(7I)) to receive the file descriptor.

If *path* names the master side of a pseudo-terminal device, then it is unspecified whether open( ) locks the slave side so that it cannot be opened. Portable applications must call unlockpt(3C) before opening the slave side.

If *path* is a symbolic link and O_CREAT and O_EXCL are set, the link is not followed.

Certain flag values can be set following open( ) as described in fcntl(2).

The largest value that can be represented correctly in an object of type off_t is established as the offset maximum in the open file description.

RETURN VALUES

Upon successful completion, the open( ) function opens the file and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, –1 is returned, errno is set to indicate the error, and no files are created or modified.

ERRORS

The open( ) function will fail if:

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

EDQUOT — The file does not exist, O_CREAT is specified, and either the directory where the new file entry is being placed cannot be extended because the user's quota of disk blocks on that file system has been exhausted, or the user's quota of inodes on the file system where the file is being created has been exhausted.

EEXIST	The O_CREAT and O_EXCL flags are set, and the named file exists.
EINTR	A signal was caught during open( ).
EFAULT	The *path* argument points to an illegal address.
EIO	The *path* argument names a STREAMS file and a hangup or error occurred during the open( ).
EISDIR	The named file is a directory and *oflag* includes O_WRONLY or O_RDWR.
ELOOP	Too many symbolic links were encountered in resolving *path*.
EMFILE	The process has too many open files. (See getrlimit(2).)
EMULTIHOP	Components of *path* require hopping to multiple remote machines and the file system does not allow it.
ENAMETOOLONG	The length of the *path* argument exceeds PATH_MAX or a pathname component is longer than NAME_MAX.
ENFILE	The maximum allowable number of files is currently open in the system.
ENOENT	The O_CREAT flag is not set and the named file does not exist; or the O_CREAT flash is set and either the path prefix does not exist or the *path* argument points to an empty string.
ENOLINK	The *path* argument points to a remote machine, and the link to that machine is no longer active.
ENOSR	The *path* argument names a STREAMS-based file and the system is unable to allocate a STREAM.
ENOSPC	The directory or file system that would contain the new file cannot be expanded, the file does not exist, and O_CREAT is specified.
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The O_NONBLOCK flag is set, the named file is a FIFO, the O_WRONLY flag is set, and no process has the file open for reading; or the named file is a character special or block special file and the device associated with this special file does not exist.

EOPNOTSUPP An attempt was made to open a path that corresponds to a AF_UNIX socket.

EOVERFLOW The named file is a regular file and either O_LARGEFILE is not set and the size of the file cannot be represented correctly in an object of type off_t or O_LARGEFILE is set and the size of the file cannot be represented correctly in an object of type off64_t.

EROFS The named file resides on a read-only file system and either O_WRONLY, O_RDWR, O_CREAT (if file does not exist), or O_TRUNC is set in the *oflag* argument.

The open( ) function may fail if:

EAGAIN The *path* argument names the slave side of a pseudo-terminal device that is locked.

EINVAL The value of the *oflag* argument is not valid.

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

ENOMEM The *path* argument names a STREAMS file and the system is unable to allocate resources.

ETXTBSY The file is a pure procedure (shared text) file that is being executed and *oflag* is O_WRONLY or O_RDWR.

USAGE

The open( ) function has a transitional interface for 64-bit file offsets. See lf64(5). Note that using open64( ) is equivalent to using open( ) with O_LARGEFILE set in *oflag*.

ATTRIBUTES

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

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

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

Trusted Solaris 8 Reference Manual

intro(2), chmod(2), creat(2), exec(2), fcntl(2), getrlimit(2), lseek(2), read(2), stat(2), write(2)

SunOS 5.8 Reference Manual

close(2), dup(2), getmsg(2), putmsg(2), umask(2), fcntl(5), stat(5), connld(7M), streamio(7I)

NOTES

Hierarchical Storage Management (HSM) file systems can sometimes cause long delays when opening a file, since HSM files must be recalled from secondary storage.

NAME

fpathconf, pathconf – Get configurable pathname variables

SYNOPSIS

#include <unistd.h>
long int **fpathconf**(int *fildes*, int *name*);

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

DESCRIPTION

The fpathconf( ) and pathconf( ) functions provide a method for the application to determine the current value of a configurable limit or option I (variable) that is associated with a file or directory.

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

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

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

Variable	Value of ***name***	Notes
FILESIZEBITS	_PC_FILESIZEBITS	3,4
LINK_MAX	_PC_LINK_MAX	1
MAX_CANON	_PC_MAX_CANON	2
MAX_INPUT	_PC_MAX_INPUT	2
NAME_MAX	_PC_NAME_MAX	3,4
PATH_MAX	_PC_PATH_MAX	4,5
PIPE_BUF	_PC_PIPE_BUF	6
_POSIX_CHOWN_RESTRICTED	_PC_CHOWN_RESTRICTED	7
_POSIX_NO_TRUNC	_PC_NO_TRUNC	3,4
_POSIX_VDISABLE	_PC_VDISABLE	2
_POSIX_ASYNC_IO	_PC_ASYNC_IO	8
_POSIX_PRIO_IO	_PC_PRIO_IO	8
_POSIX_SYNC_IO	_PC_SYNC_IO	8

Notes:

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

2. If *path* or *fildes* does not refer to a terminal file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
3. If *path* or *fildes* refers to a directory, the value returned applies to filenames within the directory.
4. If *path* or *fildes* does not refer to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.
5. If *path* or *fildes* refers to a directory, the value returned is the maximum length of a relative pathname when the specified directory is the working directory.
6. If *path* refers to a FIFO , or *fildes* refers to a pipe or FIFO , the value returned applies to the referenced object. If *path* or *fildes* refers to a directory, the value returned applies to any FIFO that exists or can be created within the directory. If *path* or *fildes* refers to any other type of file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
7. If *path* or *fildes* refers to a directory, the value returned applies to any files, other than directories, that exist or can be created within the directory.
8. If *path* or *fildes* refers to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.

RETURN VALUES

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

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

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

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

ERRORS

The `pathconf( )` function will fail if:

EINVAL	The value of *name* is not valid.
ELOOP	Too many symbolic links were encountered in resolving *path* .

The pathconf( ) function may fail if:

EACCES	Search permission is denied for a component of the path prefix.
EINVAL	The implementation does not support an association of the variable *name* with the specified file.
ENAMETOOLONG	The length of the *path* argument exceeds PATH_MAX or a pathname component is longer than NAME_MAX .
ENAMETOOLONG	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX .
ENOENT	A component of *path* does not name an existing file or *path* is an empty string.
ENOTDIR	A component of the path prefix is not a directory.

The fpathconf( ) function will fail if:

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

The fpathconf( ) function may fail if:

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

EINVAL	The implementation does not support an association of the variable *name* with the specified file.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	pathconf( ) is Async-Signal-Safe

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

SunOS 5.8 Reference Manual

sysconf(3C) , limits(4) , attributes(5) , standards(5)

NAME

p_online – Return or change processor operational status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/processor.h>
int p_online(processorid_t processorid, int flag);
```

DESCRIPTION

The p_online( ) function changes or returns the operational status of processors. The state of the processor specified by the *processorid* argument is changed to the state represented by the *flag* argument.

Legal values for *flag* are P_STATUS, P_ONLINE, P_OFFLINE, and P_NOINTR.

When *flag* is P_STATUS, no processor status change occurs, but the current processor status is returned.

The P_ONLINE, P_OFFLINE, and P_NOINTR values for *flag* refer to valid processor states. A processor in the P_ONLINE state is allowed to process LWPs (lightweight processes) and perform system activities. The processor is also interruptible by I/O devices attached to the system. The PRIV_SYS_CONFIG privilege is required.

A processor in the P_OFFLINE state is not allowed to process LWPs. The processor is as inactive as possible. If the hardware supports such a feature, the processor is not interruptible by attached I/O devices.The PRIV_SYS_CONFIG privilege is required.

A processor in the P_NOINTR state is allowed to process LWPs, but it is not interruptible by attached I/O devices. Typically, interrupts, when they occur are routed to other processors in the system. Not all systems support putting a processor into the P_NOINTR state. It is not permitted to put all the processors of a system into the P_NOINTR state. At least one processor must always be available to service system clock interrupts.

Processor numbers are integers, greater than or equal to 0, and are defined by the hardware platform. Processor numbers are not necessarily contiguous, but “not too sparse.” Processor numbers should always be printed in decimal.

The number of processors present can be determined by calling sysconf(_SC_NPROCESSORS_CONF). The list of valid processor numbers can be determined by calling p_online( ) with *processorid* values starting at 0 until all processors have been found. The EINVAL error is returned for invalid processor numbers. See EXAMPLES below.

RETURN VALUES

On successful completion, the value returned is the previous state of the processor, P_ONLINE, P_OFFLINE, P_NOINTR, or P_POWEROFF. Otherwise, –1 is returned and errno is set to indicate the error.

ERRORS

The p_online( ) function will fail if:

EPERM	The calling process does not have the PRIV_SYS_CONFIG privilege.
EINVAL	A non-existent processor ID was specified or *flag* was invalid.
EBUSY	The *flag* was P_OFFLINE and the specified processor is the only on-line processor, there are currently LWPs bound to the processor, or the processor performs some essential function that cannot be performed by another processor.
EBUSY	The *flag* was P_NOINTR and the specified processor is the only interruptible processor in the system, or it handles interrupts that cannot be handled by another processor.
EBUSY	The specified processor is powered off and cannot be powered on because some platform-specific resource is not available.
ENOTSUP	The specified processor is powered off, and the platform does not support power on of individual processors.

EXAMPLES

EXAMPLE 1 List the legal processor numbers.

The following code sample will list the legal processor numbers:

```
#include <sys/unistd.h>
#include <sys/processor.h>
#include <sys/types.h>
#include <stdio.h>
#include <errno.h>

int
main()
{
        processorid_t i;
        int     status;
        int     n = sysconf(_SC_NPROCESSORS_ONLN);
        for (i = 0; n > 0; i++) {
                status = p_online(i, P_STATUS);
                if (status == -1 && errno == EINVAL)
                        continue;
                        printf("processor %d present\n", i);
                        n--;
        }
        return (0);
}
```

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	MT-Safe

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

Trusted Solaris 8 Reference Manual

`psradm`(1M), `processor_bind`(2)

SunOS 5.8 Reference Manual

`psrinfo`(1M), `processor_info`(2), `pset_create`(2), `sysconf`(3C), `attributes`(5)

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);
```

DESCRIPTION

The `read()` function attempts to read *nbyte* bytes from the file associated with the open file descriptor, *fildes* , into the buffer pointed to by *buf* .

If *nbyte* is 0, `read()` will return `0` and have no other results.

On files that support seeking (for example, a regular file), the `read()` starts at a position in the file given by the file offset associated with *fildes* . The file offset is incremented by the number of bytes actually read.

Files that do not support seeking (for example, terminals) always read from the current position. The value of a file offset associated with such a file is undefined.

If *fildes* refers to a socket, `read()` is equivalent to `recv`(3SOCKET) with no flags set.

No data transfer will occur past the current end-of-file. If the starting position is at or after the end-of-file, `0` will be returned. If the file refers to a device special file, the result of subsequent `read()` requests is implementation-dependent.

If the value of *nbyte* is greater than `SSIZE_MAX` , the result is implementation-dependent.

When attempting to read from a regular file with mandatory file / record locking set (see `chmod`(2)), and there is a write lock owned by another process on the segment of the file to be read:

- If `O_NDELAY` or `O_NONBLOCK` is set, `read()` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read()` sleeps until the blocking record lock is removed.

When attempting to read from an empty pipe (or FIFO):

- If no process has the pipe open for writing, `read()` returns `0` to indicate end-of-file.
- If some process has the pipe open for writing and `O_NDELAY` is set, `read()` returns `0` .
- If some process has the pipe open for writing and `O_NONBLOCK` is set, `read()` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read()` blocks until data is written to the pipe or the pipe is closed by all processes that had opened the pipe for writing.

When attempting to read a file associated with a terminal that has no data currently available:

- If `O_NDELAY` is set, `read()` returns `0` .
- If `O_NONBLOCK` is set, `read()` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read()` blocks until data become available.

When attempting to read a file associated with a socket or a stream that is not a pipe, a FIFO , or a terminal, and the file has no data currently available:

- If `O_NDELAY` or `O_NONBLOCK` is set, `read()` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read()` blocks until data becomes available.

The `read()` function reads data previously written to a file. If any portion of a regular file prior to the end-of-file has not been written, `read()` returns bytes with value `0` . For example, `lseek`(2) allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return bytes with value 0 until data is written into the gap.

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with *fildes* .

Upon successful completion, where *nbyte* is greater than 0, `read()` will mark for update the `st_atime` field of the file, and return the number of bytes read. This number will never be greater than *nbyte* . The value returned may be less than *nbyte* if the number of bytes left in the file is less than *nbyte* , if the `read()` request was interrupted by a signal, or if the file is a pipe or FIFO or special file and has fewer than *nbyte* bytes immediately available for reading. For example, a `read()` from a file associated with a terminal may return one typed line of data.

If a `read()` is interrupted by a signal before it reads any data, it will return `–1` with `errno` set to `EINTR` .

If a `read( )` is interrupted by a signal after it has successfully read some data, it will return the number of bytes read.

A `read( )` or `readv( )` from a STREAMS (see `intro`(2)) file can read data in three different modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default is byte-stream mode. This can be changed using the `I_SRDOPT` `ioctl`(2) request, and can be tested with the `I_GRDOPT` `ioctl( )` . In byte-stream mode, `read( )` retrieves data from the STREAM until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, `read( )` retrieves data until as many bytes as were requested are transferred, or until a message boundary is reached. If `read( )` does not retrieve all the data in a message, the remaining data is left on the STREAM, and can be retrieved by the next `read( )` call. Message-discard mode also retrieves data until as many bytes as were requested are transferred, or a message boundary is reached. However, unread data remaining in a message after the `read( )` returns is discarded, and is not available for a subsequent `read( )` , `readv( )` or `getmsg`(2) call.

How `read( )` handles zero-byte STREAMS messages is determined by the current read mode setting. In byte-stream mode, `read( )` accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The `read( )` function then returns the number of bytes read, and places the zero-byte message back on the STREAM to be retrieved by the next `read( )` , `readv( )` or `getmsg`(2) . In message-nondiscard mode or message-discard mode, a zero-byte message returns `0` and the message is removed from the STREAM. When a zero-byte message is read as the first message on a STREAM, the message is removed from the STREAM and `0` is returned, regardless of the read mode.

A `read( )` from a STREAMS file returns the data in the message at the front of the STREAM head read queue, regardless of the priority band of the message.

By default, STREAMS are in control-normal mode, in which a `read( )` from a STREAMS file can only process messages that contain a data part but do not contain a control part. The `read( )` fails if a message containing a control part is encountered at the STREAM head. This default action can be changed by placing the STREAM in either control-data mode or control-discard mode with the `I_SRDOPT` `ioctl( )` command. In control-data mode, `read( )` converts any control part to data and passes it to the application before passing any data part originally present in the same message. In control-discard mode, `read( )` discards message control parts but returns to the process any data part in the message.

`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 filesystem object.

In all other respects, the `readl( )` , `preadl( )` , and `readvl( )` interfaces are analogous to the `read( )` , `pread( )` , and `readv( )` interfaces.

In the SunOS 5.7 operating system, `read( )` normally allows a process to read the contents of directories on some local file systems. This functionality is not supported in the Trusted Solaris operating environment. 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 filesystem object. There is no privilege to override this restriction.

In addition, `read( )` and `readv( )` will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of `errno` does not reflect the result of `read( )` or `readv( )` but reflects the prior error. If a hangup occurs on the STREAM being read, `read( )` continues to operate normally until the STREAM head read queue is empty. Thereafter, it returns `0` .

`readv( )` The `readv( )` function is equivalent to `read( )` , but places the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov* `0` , *iov* `1` , ..., *iov* [*iovcnt* –1]. The *iovcnt* argument is valid if greater than 0 and less than or equal to `IOV_MAX` .

The `iovec` structure contains the following members:

```
caddr_t   iov_base;
int       iov_len;
```

Each `iovec` entry specifies the base address and length of an area in memory where data should be placed. The `readv( )` function always fills an area completely before proceeding to the next.

Upon successful completion, `readv( )` marks for update the `st_atime` field of the file.

`pread( )` The `pread( )` function performs the same action as `read( )` , except that it reads from a given position in the file without changing the file pointer. The first three arguments to `pread( )` are the same as `read( )` with the addition of a fourth argument *offset* for the desired position inside the file. `pread( )` will read up to the maximum offset value that can be represented in an `off_t` for regular files. An attempt to perform a `pread( )` on a file that is incapable of seeking results in an error.

RETURN VALUES

Upon successful completion, `read()` and `readv()` return a non-negative integer indicating the number of bytes actually read. Otherwise, the functions return `–1` and set `errno` to indicate the error.

ERRORS

`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; or no message is waiting to be read on a stream and `O_NDELAY` or `O_NONBLOCK` was set.
`EBADF`	The *fildes* argument is not a valid file descriptor open for reading.
`EBADMSG`	Message waiting to be read on a stream is not a data message.
`EDEADLK`	The read was going to go to sleep and cause a deadlock to occur.
`EFAULT`	The *buf* argument points to an illegal address.
`EINTR`	A signal was caught during the read operation and no data was transferred.
`EINVAL`	An attempt was made to read from a stream linked to a multiplexor.
`EIO`	A physical I/O error has occurred, or the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the `SIGTTIN` signal or the process group of the process is orphaned.
`EISDIR`	The *fildes* argument refers to a directory on a file system type that does not support read operations on directories.
`ENOLCK`	The system record lock table was full, so the `read()` or `readv()` could not go to sleep until the blocking record lock was removed.
`ENOLINK`	The *fildes* argument is on a remote machine and the link to that machine is no longer active.

ENXIO The device associated with *fildes* is a block special or character special file and the value of the file pointer is out of range.

The `read( )` and `readv( )` functions will fail if:

EOVERFLOW The file is a regular file, *nbyte* is greater than 0, the starting position is before the end-of-file, and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fildes* .

The `readv( )` function may fail if:

EFAULT The *iov* argument points outside the allocated address space.

EINVAL The *iovcnt* argument was less than or equal to `0` , or greater than or equal to `IOV_MAX` . (See `intro`(2) for a definition of `IOV_MAX`).

EINVAL The sum of the `iov_len` values in the *iov* array overflowed an int.

The `pread( )` function will fail and the file pointer remain unchanged if:

EFAULT *label_p* points to an illegal address.

USAGE

The `pread( )` function has a transitional interface for 64-bit file offsets. See `lf64`(5) .

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	`read( )` is Async-Signal-Safe

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 filesystem object. There is no privilege to override this restriction.

For conduits, a sensitivity label is associated with each byte of data.

SEE ALSO

Trusted Solaris 8 Reference Manual

`intro`(2) , `chmod`(2) , `creat`(2) , `fcntl`(2) , `open`(2)

SunOS 5.8 Reference Manual

`dup`(2) , `getmsg`(2) , `ioctl`(2) , `pipe`(2) , `streamio`(7I) , `termio`(7I)

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);
```

DESCRIPTION

The `read()` function attempts to read *nbyte* bytes from the file associated with the open file descriptor, *fildes* , into the buffer pointed to by *buf* .

If *nbyte* is 0, `read()` will return `0` and have no other results.

On files that support seeking (for example, a regular file), the `read()` starts at a position in the file given by the file offset associated with *fildes* . The file offset is incremented by the number of bytes actually read.

Files that do not support seeking (for example, terminals) always read from the current position. The value of a file offset associated with such a file is undefined.

If *fildes* refers to a socket, `read()` is equivalent to `recv`(3SOCKET) with no flags set.

No data transfer will occur past the current end-of-file. If the starting position is at or after the end-of-file, `0` will be returned. If the file refers to a device special file, the result of subsequent `read()` requests is implementation-dependent.

If the value of *nbyte* is greater than `SSIZE_MAX` , the result is implementation-dependent.

When attempting to read from a regular file with mandatory file / record locking set (see `chmod`(2)), and there is a write lock owned by another process on the segment of the file to be read:

- If `O_NDELAY` or `O_NONBLOCK` is set, `read()` returns –`1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read()` sleeps until the blocking record lock is removed.

When attempting to read from an empty pipe (or FIFO):

- If no process has the pipe open for writing, `read( )` returns `0` to indicate end-of-file.
- If some process has the pipe open for writing and `O_NDELAY` is set, `read( )` returns `0` .
- If some process has the pipe open for writing and `O_NONBLOCK` is set, `read( )` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read( )` blocks until data is written to the pipe or the pipe is closed by all processes that had opened the pipe for writing.

When attempting to read a file associated with a terminal that has no data currently available:

- If `O_NDELAY` is set, `read( )` returns `0` .
- If `O_NONBLOCK` is set, `read( )` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read( )` blocks until data become available.

When attempting to read a file associated with a socket or a stream that is not a pipe, a FIFO , or a terminal, and the file has no data currently available:

- If `O_NDELAY` or `O_NONBLOCK` is set, `read( )` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read( )` blocks until data becomes available.

The `read( )` function reads data previously written to a file. If any portion of a regular file prior to the end-of-file has not been written, `read( )` returns bytes with value `0` . For example, `lseek`(2) allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return bytes with value 0 until data is written into the gap.

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with *fildes* .

Upon successful completion, where *nbyte* is greater than 0, `read( )` will mark for update the `st_atime` field of the file, and return the number of bytes read. This number will never be greater than *nbyte* . The value returned may be less than *nbyte* if the number of bytes left in the file is less than *nbyte* , if the `read( )` request was interrupted by a signal, or if the file is a pipe or FIFO or special file and has fewer than *nbyte* bytes immediately available for reading. For example, a `read( )` from a file associated with a terminal may return one typed line of data.

If a `read( )` is interrupted by a signal before it reads any data, it will return `–1` with `errno` set to `EINTR` .

If a `read( )` is interrupted by a signal after it has successfully read some data, it will return the number of bytes read.

A `read( )` or `readv( )` from a STREAMS (see `intro`(2)) file can read data in three different modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default is byte-stream mode. This can be changed using the `I_SRDOPT` `ioctl`(2) request, and can be tested with the `I_GRDOPT` `ioctl( )` . In byte-stream mode, `read( )` retrieves data from the STREAM until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, `read( )` retrieves data until as many bytes as were requested are transferred, or until a message boundary is reached. If `read( )` does not retrieve all the data in a message, the remaining data is left on the STREAM, and can be retrieved by the next `read( )` call. Message-discard mode also retrieves data until as many bytes as were requested are transferred, or a message boundary is reached. However, unread data remaining in a message after the `read( )` returns is discarded, and is not available for a subsequent `read( )` , `readv( )` or `getmsg`(2) call.

How `read( )` handles zero-byte STREAMS messages is determined by the current read mode setting. In byte-stream mode, `read( )` accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The `read( )` function then returns the number of bytes read, and places the zero-byte message back on the STREAM to be retrieved by the next `read( )` , `readv( )` or `getmsg`(2) . In message-nondiscard mode or message-discard mode, a zero-byte message returns `0` and the message is removed from the STREAM. When a zero-byte message is read as the first message on a STREAM, the message is removed from the STREAM and `0` is returned, regardless of the read mode.

A `read( )` from a STREAMS file returns the data in the message at the front of the STREAM head read queue, regardless of the priority band of the message.

By default, STREAMS are in control-normal mode, in which a `read( )` from a STREAMS file can only process messages that contain a data part but do not contain a control part. The `read( )` fails if a message containing a control part is encountered at the STREAM head. This default action can be changed by placing the STREAM in either control-data mode or control-discard mode with the `I_SRDOPT` `ioctl( )` command. In control-data mode, `read( )` converts any control part to data and passes it to the application before passing any data part originally present in the same message. In control-discard mode, `read( )` discards message control parts but returns to the process any data part in the message.

`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 filesystem object.

In all other respects, the `readl( )` , `preadl( )` , and `readvl( )` interfaces are analogous to the `read( )` , `pread( )` , and `readv( )` interfaces.

In the SunOS 5.7 operating system, `read( )` normally allows a process to read the contents of directories on some local file systems. This functionality is not supported in the Trusted Solaris operating environment. 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 filesystem object. There is no privilege to override this restriction.

In addition, `read( )` and `readv( )` will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of `errno` does not reflect the result of `read( )` or `readv( )` but reflects the prior error. If a hangup occurs on the STREAM being read, `read( )` continues to operate normally until the STREAM head read queue is empty. Thereafter, it returns `0` .

`readv( )` The `readv( )` function is equivalent to `read( )` , but places the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov* `0` , *iov* `1` , ..., *iov* [*iovcnt* –1]. The *iovcnt* argument is valid if greater than 0 and less than or equal to `IOV_MAX` .

The `iovec` structure contains the following members:

```
caddr_t   iov_base;
int       iov_len;
```

Each `iovec` entry specifies the base address and length of an area in memory where data should be placed. The `readv( )` function always fills an area completely before proceeding to the next.

Upon successful completion, `readv( )` marks for update the `st_atime` field of the file.

`pread( )` The `pread( )` function performs the same action as `read( )` , except that it reads from a given position in the file without changing the file pointer. The first three arguments to `pread( )` are the same as `read( )` with the addition of a fourth argument *offset* for the desired position inside the file. `pread( )` will read up to the maximum offset value that can be represented in an `off_t` for regular files. An attempt to perform a `pread( )` on a file that is incapable of seeking results in an error.

RETURN VALUES

Upon successful completion, `read()` and `readv()` return a non-negative integer indicating the number of bytes actually read. Otherwise, the functions return `–1` and set `errno` to indicate the error.

ERRORS

`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; or no message is waiting to be read on a stream and `O_NDELAY` or `O_NONBLOCK` was set.
`EBADF`	The *fildes* argument is not a valid file descriptor open for reading.
`EBADMSG`	Message waiting to be read on a stream is not a data message.
`EDEADLK`	The read was going to go to sleep and cause a deadlock to occur.
`EFAULT`	The *buf* argument points to an illegal address.
`EINTR`	A signal was caught during the read operation and no data was transferred.
`EINVAL`	An attempt was made to read from a stream linked to a multiplexor.
`EIO`	A physical I/O error has occurred, or the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the `SIGTTIN` signal or the process group of the process is orphaned.
`EISDIR`	The *fildes* argument refers to a directory on a file system type that does not support read operations on directories.
`ENOLCK`	The system record lock table was full, so the `read()` or `readv()` could not go to sleep until the blocking record lock was removed.
`ENOLINK`	The *fildes* argument is on a remote machine and the link to that machine is no longer active.

ENXIO — The device associated with *fildes* is a block special or character special file and the value of the file pointer is out of range.

The `read()` and `readv()` functions will fail if:

EOVERFLOW — The file is a regular file, *nbyte* is greater than 0, the starting position is before the end-of-file, and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fildes* .

The `readv()` function may fail if:

EFAULT — The *iov* argument points outside the allocated address space.

EINVAL — The *iovcnt* argument was less than or equal to `0` , or greater than or equal to `IOV_MAX` . (See `intro`(2) for a definition of `IOV_MAX`).

EINVAL — The sum of the `iov_len` values in the *iov* array overflowed an int.

The `pread()` function will fail and the file pointer remain unchanged if:

EFAULT — *label_p* points to an illegal address.

USAGE

The `pread()` function has a transitional interface for 64-bit file offsets. See `lf64`(5) .

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	`read()` is Async-Signal-Safe

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 filesystem object. There is no privilege to override this restriction.

For conduits, a sensitivity label is associated with each byte of data.

SEE ALSO

Trusted Solaris 8 Reference Manual

`intro`(2) , `chmod`(2) , `creat`(2) , `fcntl`(2) , `open`(2)

SunOS 5.8 Reference Manual

`dup`(2) , `getmsg`(2) , `ioctl`(2) , `pipe`(2) , `streamio`(7I) , `termio`(7I)

NAME

priocntl – Process scheduler control

SYNOPSIS

```
#include <sys/types.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>
long priocntl(idtype_t idtype, id_t id, int cmd, /* arg */ ...);
```

DESCRIPTION

The `priocntl( )` function provides for control over the scheduling of an active light weight process (LWP).

LWPs fall into distinct classes with a separate scheduling policy applied to each class. The two classes currently supported are the realtime class and the time-sharing class. The characteristics of these classes are described under the corresponding headings below. The class attribute of an LWP is inherited across the `fork`(2) and `_lwp_create`(2) functions and the *exec* family of functions (see `exec`(2)). The `priocntl( )` function can be used to dynamically change the class and other scheduling parameters associated with a running LWP or set of LWPs given the appropriate permissions as explained below.

In the default configuration, a runnable realtime LWP runs before any other LWP. Therefore, inappropriate use of realtime LWP can have a dramatic negative impact on system performance.

The `priocntl( )` function provides an interface for specifying a process, set of processes or an LWP to which the function is to apply. The `priocntlset`(2) function provides the same functions as `priocntl( )`, but allows a more general interface for specifying the set of LWPs to which the function is to apply.

For `priocntl( )`, the *idtype* and *id* arguments are used together to specify the set of LWPs. The interpretation of *id* depends on the value of *idtype*. The possible values for *idtype* and corresponding interpretations of *id* are as follows:

`P_LWPID` The *id* argument is an LWP ID. The *priocntl* function applies to the LWP with the specified ID within the calling process.

`P_PID` The *id* argument is a process ID specifying a single process. The `priocntl( )` function applies to all LWPs currently associated with the specified process.

`P_PPID` The *id* argument is a parent process ID. The `priocntl( )` function applies to all LWPs currently associated with processes with the specified parent process ID.

`P_PGID` The *id* argument is a process group ID. The `priocntl( )` function applies to all LWPs currently associated with processes in the specified process group.

P_SID	The *id* argument is a session ID. The priocntl( ) function applies to all LWPs currently associated with processes in the specified session.
P_CID	The *id* argument is a class ID (returned by the priocntl( ) PC_GETCID command as explained below). The priocntl( ) function applies to all LWPs in the specified class.
P_UID	The *id* argument is a user ID. The priocntl( ) function applies to all LWPs with this effective user ID.
P_GID	The *id* argument is a group ID. The priocntl( ) function applies to all LWPs with this effective group ID.
P_ALL	The priocntl( ) function applies to all existing LWPs. The value of *id* is ignored. The permission restrictions described below still apply.

An *id* value of P_MYID can be used in conjunction with the *idtype* value to specify the calling LWP's LWP ID, parent process ID, process group ID, session ID, 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 0 would specify all processes with a user ID of 0 except processes in the sys class and (if changing the parameters using PC_SETPARMS) the init(1M) process.

The *init* process is a special case. In order for a priocntl( ) call to change the class or other scheduling parameters of the *init* process (process ID 1), it must be the only process specified by *idtype* and *id*. The *init* process may be assigned to any class configured on the system, but the time-sharing class is almost always the appropriate choice. (Other choices may be highly undesirable; see the *System Administration Guide, Volume 1* for more information.)

The data type and value of *arg* are specific to the type of command specified by *cmd*.

A structure with the following members is used by the PC_GETCID and PC_GETCLINFO commands.

```
id_t   pc_cid;                  /* Class id */
char   pc_clname[PC_CLNMSZ];    /* Class name */
int    pc_clinfo[PC_CLINFOSZ];  /* Class information */
```

The pc_cid member is a class ID returned by the priocntl( ) PC_GETCID command. The pc_clname member is a buffer of size PC_CLNMSZ (defined in <sys/priocntl.h>) used to hold the class name (RT for realtime or TS for time-sharing).

The pc_clinfo member is a buffer of size PC_CLINFOSZ (defined in <sys/priocntl.h>) used to return data describing the attributes of a specific class. The format of this data is class-specific and is described under the appropriate heading (REALTIME CLASS or TIME-SHARING CLASS) below.

A structure with the following elements is used by the PC_SETPARMS and PC_GETPARMS commands.

```
id_t  pc_cid;                  /* LWP class */
int   pc_clparms[PC_CLPARMSZ];  /* Class-specific params */
```

The pc_cid member is a class ID (returned by the priocntl( ) PC_GETCID command). The special class ID PC_CLNULL can also be assigned to pc_cid when using the PC_GETPARMS command as explained below.

The pc_clparms buffer holds class-specific scheduling parameters. The format of this parameter data for a particular class is described under the appropriate heading below. PC_CLPARMSZ is the length of the pc_clparms buffer and is defined in <sys/priocntl.h>.

COMMANDS

Available priocntl( ) commands are:

PC_GETCID

Get class ID and class attributes for a specific class given class name. The *idtype* and *id* arguments are ignored. If *arg* is non-null, it points to a structure of type pcinfo_t. The pc_clname buffer contains the name of the class whose attributes you are getting.

On success, the class ID is returned in pc_cid, the class attributes are returned in the pc_clinfo buffer, and the priocntl( ) call returns the total number of classes configured in the system (including the sys

class). If the class specified by pc_clname is invalid or is not currently configured the priocntl( ) call returns –1 with errno set to EINVAL. The format of the attribute data returned for a given class is defined in the <sys/rtpriocntl.h> or <sys/tspriocntl.h> header and described under the appropriate heading below.

If *arg* is a null pointer, no attribute data is returned but the priocntl( ) call still returns the number of configured classes.

PC_GETCLINFO
Get class name and class attributes for a specific class given class ID. The *idtype* and *id* arguments are ignored. If *arg* is non-null, it points to a structure of type pcinfo_t. The pc_cid member is the class ID of the class whose attributes you are getting.

On success, the class name is returned in the pc_clname buffer, the class attributes are returned in the pc_clinfo buffer, and the priocntl( ) call returns the total number of classes configured in the system (including the sys class). The format of the attribute data returned for a given class is defined in the <sys/rtpriocntl.h> or <sys/tspriocntl.h> header file and described under the appropriate heading below.

If *arg* is a null pointer, no attribute data is returned but the priocntl( ) call still returns the number of configured classes.

PC_SETPARMS
Set the class and class-specific scheduling parameters of the specified LWP(s) associated with the specified process(es). When this command is used with the *idtype* of P_LWPID, it will set the class and class-specific scheduling parameters of the LWP. The *arg* argument points to a structure of type pcparms_t. The pc_cid member specifies the class you are setting and the pc_clparms buffer contains the class-specific parameters you are setting. The format of the class-specific parameter data is defined in the <sys/rtpriocntl.h> or <sys/tspriocntl.h> header and described under the appropriate class heading below.

When setting parameters for a set of LWPs, priocntl( ) acts on the LWPs in the set in an implementation-specific order. If priocntl( ) encounters an error for one or more of the target processes, it may or may not continue through the set of LWPs, depending on the nature of the error. If the error is related to permissions (EPERM), priocntl( ) continues through the LWP set, resetting the parameters for all target LWPs for which the calling LWP has appropriate permissions. The priocntl( ) function then returns –1 with errno set to EPERM to indicate that the operation failed for one or more of the target LWPs. If priocntl( ) encounters an error other

than permissions, it does not continue through the set of target LWPs but returns the error immediately.

PC_GETPARMS
Get the class and/or class-specific scheduling parameters of an LWP. The *arg* member points to a structure of type pcparms_t.

If pc_cid specifies a configured class and a single LWP belonging to that class is specified by the *idtype* and *id* values or the procset structure, then the scheduling parameters of that LWP are returned in the pc_clparms buffer. If the LWP specified does not exist or does not belong to the specified class, the priocntl( ) call returns –1 with errno set to ESRCH.

If pc_cid specifies a configured class and a set of LWPs is specified, the scheduling parameters of one of the specified LWP belonging to the specified class are returned in the pc_clparms buffer and the priocntl( ) call returns the process ID of the selected LWP. The criteria for selecting an LWP to return in this case is class dependent. If none of the specified LWPs exist or none of them belong to the specified class the priocntl( ) call returns –1 with errno set to ESRCH.

If pc_cid is PC_CLNULL and a single LWP is specified the class of the specified LWP is returned in pc_cid and its scheduling parameters are returned in the pc_clparms buffer.

PC_ADMIN
This command provides functionality needed for the implementation of the dispadmin(1) utility. It is not intended for general use by other applications.

REALTIME CLASS

The realtime class provides a fixed priority preemptive scheduling policy for those LWPs requiring fast and deterministic response and absolute user/application control of scheduling priorities. If the realtime class is configured in the system it should have exclusive control of the highest range of scheduling priorities on the system. This ensures that a runnable realtime LWP is given CPU service before any LWP belonging to any other class.

The realtime class has a range of realtime priority (rt_pri) values that may be assigned to an LWP within the class. Real-time priorities range from 0 to *x*, where the value of *x* is configurable and can be determined for a specific installation by using the priocntl( ) PC_GETCID or PC_GETCLINFO command.

The realtime scheduling policy is a fixed priority policy. The scheduling priority of a realtime LWP is never changed except as the result of an explicit request by the user/application to change the rt_pri value of the LWP.

For an LWP in the realtime class, the rt_pri value is, for all practical purposes, equivalent to the scheduling priority of the LWP. The rt_pri value completely

determines the scheduling priority of a realtime LWP relative to other LWPs within its class. Numerically higher `rt_pri` values represent higher priorities. Since the realtime class controls the highest range of scheduling priorities in the system it is guaranteed that the runnable realtime LWP with the highest `rt_pri` value is always selected to run before any other LWPs in the system.

In addition to providing control over priority, `priocntl( )` provides for control over the length of the time quantum allotted to the LWP in the realtime class. The time quantum value specifies the maximum amount of time an LWP may run assuming that it does not complete or enter a resource or event wait state (*sleep*). Note that if another LWP becomes runnable at a higher priority, the currently running LWP may be preempted before receiving its full time quantum.

The system's process scheduler keeps the runnable realtime LWPs on a set of scheduling queues. There is a separate queue for each configured realtime priority and all realtime LWPs with a given `rt_pri` value are kept together on the appropriate queue. The LWPs on a given queue are ordered in FIFO order (that is, the LWP at the front of the queue has been waiting longest for service and receives the CPU first). Real-time LWPs that wake up after sleeping, LWPs which change to the realtime class from some other class, LWPs which have used their full time quantum, and runnable LWPs whose priority is reset by `priocntl( )` are all placed at the back of the appropriate queue for their priority. An LWP that is preempted by a higher priority LWP remains at the front of the queue (with whatever time is remaining in its time quantum) and runs before any other LWP at this priority. Following a `fork`(2) or `_lwp_create`(2) function call by a realtime LWP, the parent LWP continues to run while the child LWP (which inherits its parent's `rt_pri` value) is placed at the back of the queue.

A structure with the following members (defined in `<sys/rtpriocntl.h>`) defines the format used for the attribute data for the realtime class.

```
short    rt_maxpri;      /* Maximum realtime priority */
```

The `priocntl( )` `PC_GETCID` and `PC_GETCLINFO` commands return realtime class attributes in the `pc_clinfo` buffer in this format.

The `rt_maxpri` member specifies the configured maximum `rt_pri` value for the realtime class (if `rt_maxpri` is *x*, the valid realtime priorities range from 0 to *x*).

A structure with the following members (defined in `<sys/rtpriocntl.h>`) defines the format used to specify the realtime class-specific scheduling parameters of an LWP.

```
short    rt_pri;       /* Real-Time priority */
uint_t   rt_tqsecs;    /* Seconds in time quantum */
int      rt_tqnsecs;   /* Additional nanoseconds in quantum */
```

When using the priocntl( ) PC_SETPARMS or PC_GETPARMS commands, if pc_cid specifies the realtime class, the data in the pc_clparms buffer is in this format.

The above commands can be used to set the realtime priority to the specified value or get the current rt_pri value. Setting the rt_pri value of an LWP that is currently running or runnable (not sleeping) causes the LWP to be placed at the back of the scheduling queue for the specified priority. The LWP is placed at the back of the appropriate queue regardless of whether the priority being set is different from the previous rt_pri value of the LWP. Note that a running LWP can voluntarily release the CPU and go to the back of the scheduling queue at the same priority by resetting its rt_pri value to its current realtime priority value. In order to change the time quantum of an LWP without setting the priority or affecting the LWP's position on the queue, the rt_pri member should be set to the special value RT_NOCHANGE (defined in <sys/rtpriocntl.h>>). Specifying RT_NOCHANGE when changing the class of an LWP to realtime from some other class results in the realtime priority being set to 0.

For the priocntl( ) PC_GETPARMS command, if pc_cid specifies the realtime class and more than one realtime LWP is specified, the scheduling parameters of the realtime LWP with the highest rt_pri value among the specified LWPs are returned and the LWP ID of this LWP is returned by the priocntl( ) call. If there is more than one LWP sharing the highest priority, the one returned is implementation-dependent.

The rt_tqsecs and rt_tqnsecs members are used for getting or setting the time quantum associated with an LWP or group of LWPs. rt_tqsecs is the number of seconds in the time quantum and rt_tqnsecs is the number of additional nanoseconds in the quantum. For example setting rt_tqsecs to 2 and rt_tqnsecs to 500,000,000 (decimal) would result in a time quantum of two and one-half seconds. Specifying a value of 1,000,000,000 or greater in the rt_tqnsecs member results in an error return with errno set to EINVAL. Although the resolution of the tq_nsecs member is very fine, the specified time quantum length is rounded up by the system to the next integral multiple of the system clock's resolution. The maximum time quantum that can be specified is implementation-specific and equal to LONG_MAX1 ticks (defined in <limits.h>). Requesting a quantum greater than this maximum results in an error return with errno set to ERANGE (although infinite quantums may be requested using a special value as explained below). Requesting a time quantum

of 0 (setting both `rt_tqsecs` and `rt_tqnsecs` to 0) results in an error return with `errno` set to `EINVAL`.

The `rt_tqnsecs` member can also be set to one of the following special values (defined in <`sys/rtpriocntl.h`>), in which case the value of `rt_tqsecs` is ignored:

`RT_TQINF` Set an infinite time quantum.

`RT_TQDEF` Set the time quantum to the default for this priority (see `rt_dptbl`(4)).

`RT_NOCHANGE` Do not set the time quantum. This value is useful when you wish to change the realtime priority of an LWP without affecting the time quantum. Specifying this value when changing the class of an LWP to realtime from some other class is equivalent to specifying `RT_TQDEF`.

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`(2) and `exec`(2) system calls.

TIME-SHARING CLASS

The time-sharing scheduling policy provides for a fair and effective allocation of the CPU resource among LWPs with varying CPU consumption characteristics. The objectives of the time-sharing policy are to provide good response time to interactive LWPs and good throughput to CPU-bound jobs while providing a degree of user / application control over scheduling.

The time-sharing class has a range of time-sharing user priority (see `ts_upri` below) values that may be assigned to LWPs within the class. A `ts_upri` value of 0 is defined as the default base priority for the time-sharing class. User priorities range from –`x` to +*x* where the value of *x* is configurable and can be determined for a specific installation by using the `priocntl( )` `PC_GETCID` or `PC_GETCLINFO` command.

The purpose of the user priority is to provide some degree of user / application control over the scheduling of LWPs in the time-sharing class. Raising or lowering the `ts_upri` value of an LWP in the time-sharing class raises or lowers the scheduling priority of the LWP. It is not guaranteed, however, that an LWP with a higher `ts_upri` value will run before one with a lower `ts_upri` value. This is because the `ts_upri` value is just one factor used to determine the scheduling priority of a time-sharing LWP. The system may dynamically adjust

the internal scheduling priority of a time-sharing LWP based on other factors such as recent CPU usage.

In addition to the system-wide limits on user priority (returned by the PC_GETCID and PC_GETCLINFO commands) there is a per LWP user priority limit (see ts_uprilim below), which specifies the maximum ts_upri value that may be set for a given LWP; by default, ts_uprilim is 0.

A structure with the following members (defined in <sys/tspriocntl.h>) defines the format used for the attribute data for the time-sharing class.

```
short    ts_maxupri;     /* Limits of user priority range */
```

The priocntl( ) PC_GETCID and PC_GETCLINFO commands return time-sharing class attributes in the pc_clinfo buffer in this format.

ts_maxupri specifies the configured maximum user priority value for the time-sharing class. If ts_maxupri is *x*, the valid range for both user priorities and user priority limits is from –x to +*x*.

A structure with the following members (defined in <sys/tspriocntl.h>) defines the format used to specify the time-sharing class-specific scheduling parameters of an LWP.

```
short    ts_uprilim;     /* Time-Sharing user priority limit */
short    ts_upri;        /* Time-Sharing user priority */
```

When using the priocntl( ) PC_SETPARMS or PC_GETPARMS commands, if pc_cid specifies the time-sharing class, the data in the pc_clparms buffer is in this format.

For the priocntl( ) PC_GETPARMS command, if pc_cid specifies the time-sharing class and more than one time-sharing LWP is specified, the scheduling parameters of the time-sharing LWP with the highest ts_upri value among the specified LWPs is returned and the LWP ID of this LWP is returned by the priocntl( ) call. If there is more than one LWP sharing the highest user priority, the one returned is implementation-dependent.

Any time-sharing LWP may lower its own ts_uprilim (or that of another LWP with the same user ID). Only a time-sharing LWP with 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 members may be set to the special value TS_NOCHANGE (defined in <sys/tspriocntl.h>) in order to set one of the values without affecting the other. Specifying TS_NOCHANGE for the ts_upri when the ts_uprilim is being set to a value below the current ts_upri causes the ts_upri to be set equal to the ts_uprilim being set. Specifying TS_NOCHANGE for a parameter when changing the class of an LWP to time-sharing (from some other class) causes the parameter to be set to a default value. The default value for the ts_uprilim is 0 and the default for the ts_upri is to set it equal to the ts_uprilim which is being set.

The time-sharing user priority and user priority limit are inherited across fork( ) and the *exec* family of functions.

RETURN VALUES

Unless otherwise noted above, priocntl( ) returns a value of 0 on success. On failure, priocntl( ) returns –1 and sets errno to indicate the error.

ERRORS

The priocntl( ) function fails if:

EAGAIN	An attempt to change the class of an LWP failed because of insufficient resources other than memory (for example, class-specific kernel data structures).
EFAULT	One of the arguments points to an illegal address.
EINVAL	The argument *cmd* was invalid, an invalid or unconfigured class was specified, or one of the parameters specified was invalid.
ENOMEM	An attempt to change the class of an LWP failed because of insufficient memory.
EPERM	The calling LWP does not have required privileges.
ERANGE	The requested time quantum is out of range.
ESRCH	None of the specified LWPs exist.

SUMMARY OF TRUSTED SOLARIS CHANGES

The Trusted Solaris environment replaces the checks of super-user in the Solaris environment with privilege checks. MAC policy is enforced in addition to DAC.

SEE ALSO

Trusted Solaris 8 Reference Manual

`dispadmin`(1M), `exec`(2), `fork`(2), `nice`(2), `priocntlset`(2)

SunOS 5.8 Reference Manual

`priocntl`(1), `init`(1M), `_lwp_create`(2), `rt_dptbl`(4)

System Administration Guide, Volume 1 System Interface Guide

NAME

priocntlset – Generalized process scheduler control

SYNOPSIS

```
#include <sys/types.h>
#include <sys/procset.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>
long priocntlset(procset_t *psp, int cmd, /* arg */ ...);
```

DESCRIPTION

The `priocntlset()` function changes the scheduling properties of running processes. `priocntlset()` has the same functions as the `priocntl()` function, but a more general way of specifying the set of processes whose scheduling properties are to be changed.

cmd specifies the function to be performed. *arg* is a pointer to a structure whose type depends on *cmd*. See `priocntl`(2) for the valid values of *cmd* and the corresponding *arg* structures.

psp is a pointer to a `procset` structure, which `priocntlset()` uses to specify the set of processes whose scheduling properties are to be changed. The `procset` structure contains the following members:

```
idop_t    p_op;        /* operator connecting left/right sets */
idtype_t  p_lidtype;   /* left set ID type */
id_t      p_lid;       /* left set ID */
idtype_t  p_ridtype;   /* right set ID type */
id_t      p_rid;       /* right set ID */
```

The `p_lidtype` and `p_lid` members specify the ID type and ID of one (“left”) set of processes; the `p_ridtype` and `p_rid` members specify the ID type and ID of a second (“right”) set of processes. ID types and IDs are specified just as for the `priocntl()` function. The `p_op` member specifies the operation to be performed on the two sets of processes to get the set of processes the function is to apply to. The valid values for `p_op` and the processes they specify are:

`POP_DIFF` Set difference: processes in left set and not in right set.

`POP_AND` Set intersection: processes in both left and right sets.

`POP_OR` Set union: processes in either left or right sets or both.

`POP_XOR` Set exclusive-or: processes in left or right set but not in both.

The following macro, which is defined in `<procset.h>`, offers a convenient way to initialize a `procset` structure:

```
#define  setprocset(psp, op, ltype, lid, rtype, rid) \
(psp)⇒p_op       = (op), \
(psp)⇒p_lidtype  = (ltype), \
(psp)⇒p_lid      = (lid), \
```

```
(psp)⇒p_ridtype  = (rtype), \
(psp)⇒p_rid      = (rid),
```

RETURN VALUES

Unless otherwise noted above, `priocntlset()` returns `0` on success. Otherwise, it returns `−1` and sets `errno` to indicate the error.

ERRORS

The `priocntlset()` function will fail if:

`EAGAIN`	An attempt to change the class of a process failed because of insufficient resources other than memory (for example, class-specific kernel data structures).
`EFAULT`	One of the arguments points to an illegal address.
`EINVAL`	The argument *cmd* was invalid, an invalid or unconfigured class was specified, or one of the parameters specified was invalid.
`ENOMEM`	An attempt to change the class of a process failed because of insufficient memory.
`EPERM`	The calling process does not have required privileges.
`ERANGE`	The requested time quantum is out of range.
`ESRCH`	None of the specified processes exist.

SUMMARY OF TRUSTED SOLARIS CHANGES

The Trusted Solaris environment replaces the checks of super-user in the Solaris environment with privilege checks.

SEE ALSO

Trusted Solaris 8 Reference Manual

`priocntl`(2)

SunOS 5.8 Reference Manual

`priocntl`(1)

NAME

processor_bind – Bind LWPs to a processor

SYNOPSIS

```
#include <sys/types.h>
#include <sys/processor.h>
#include <sys/procset.h>
int processor_bind(idtype_t idtype, id_t id, processorid_t processorid, processorid_t
*obind);
```

DESCRIPTION

The `processor_bind()` function binds the LWP (lightweight process) or set of LWPs specified by *idtype* and *id* to the processor specified by *processorid*. If *obind* is not `NULL`, this function also sets the `processorid_t` variable pointed to by *obind* to the previous binding of one of the specified LWPs, or to `PBIND_NONE` if the selected LWP was not bound.

If *idtype* is `P_PID`, the binding 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

Upon successful completion, `0` is returned. Otherwise, `–1` is returned and `errno` is set to indicate the error.

ERRORS

The `processor_bind()` function will fail if:

`ESRCH` No processes or LWPs were found to match the criteria specified by *idtype* and *id*.

`EINVAL` The specified processor is not on-line, or the *idtype* argument 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 Trusted Solaris environment replaces the checks of superuser in the Solaris environment with privilege checks.

SEE ALSO

Trusted Solaris 8 Reference Manual

`psradm`(1M), `p_online`(2)

SunOS 5.8 Reference Manual

`psrinfo`(1M), `pset_bind`(2), `sysconf`(3C)

NAME

write, pwrite, writev, writel, pwritel, writevl – Write on a file

SYNOPSIS

```
#include <unistd.h>
ssize_t write(int fildes, const void * buf, size_t nbyte);

ssize_t pwrite(int fildes, const void * buf, size_t nbyte, off_t offset);

#include <sys/uio.h>
ssize_t writev(int fildes, const struct iovec * iov, int iovcnt);

#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);
```

DESCRIPTION

The `write()` function attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor, *fildes* .

If *nbyte* is `0` , `write()` will return `0` and have no other results if the file is a regular file; otherwise, the results are unspecified.

On a regular file or other file capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file offset associated with *fildes* . Before successful return from `write()` , the file offset is incremented by the number of bytes actually written. On a regular file, if this incremented file offset is greater than the length of the file, the length of the file will be set to this file offset.

If the `O_SYNC` flag of the file status flags is set and *fildes* refers to a regular file, a successful `write()` does not return until the data is delivered to the underlying hardware.

If *fildes* refers to a socket, `write()` is equivalent to `send`(3SOCKET) with no flags set.

On a file not capable of seeking, writing always takes place starting at the current position. The value of a file offset associated with such a device is undefined.

If the `O_APPEND` flag of the file status flags is set, the file offset will be set to the end of the file prior to each write and no intervening file modification operation will occur between changing the file offset and the write operation.

For regular files, no data transfer will occur past the offset maximum established in the open file description with *fildes* .

A `write()` to a regular file is blocked if mandatory file / record locking is set (see `chmod`(2)), and there is a record lock owned by another process on the segment of the file to be written:

- If O_NDELAY or O_NONBLOCK is set, write( ) returns –1 and sets errno to EAGAIN .
- If O_NDELAY and O_NONBLOCK are clear, write( ) sleeps until all blocking locks are removed or the write( ) is terminated by a signal.

If a write( ) requests that more bytes be written than there is room for—for example, if the write would exceed the process file size limit (see getrlimit(2) and ulimit(2)), the system file size limit, or the free space on the device—only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write( ) of 512-bytes returns 20 . The next write( ) of a non-zero number of bytes gives a failure return (except as noted for pipes and FIFO below).

If write( ) is interrupted by a signal before it writes any data, it will return –1 with errno set to EINTR .

If write( ) is interrupted by a signal after it successfully writes some data, it will return the number of bytes written.

If the value of *nbyte* is greater than SSIZE_MAX , the result is implementation-dependent.

After a write( ) to a regular file has successfully returned:

- Any successful read(2) from each byte position in the file that was modified by that write will return the data specified by the write( ) for that position until such byte positions are again modified.
- Any subsequent successful write( ) to the same byte position in the file will overwrite that file data.

Write requests to a pipe or FIFO are handled the same as a regular file with the following exceptions:

- There is no file offset associated with a pipe, hence each write request appends to the end of the pipe.
- Write requests of {PIPE_BUF} bytes or less are guaranteed not to be interleaved with data from other processes doing writes on the same pipe. Writes of greater than {PIPE_BUF} bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the O_NONBLOCK or O_NDELAY flags are set.
- If O_NONBLOCK and O_NDELAY are clear, a write request may cause the process to block, but on normal completion it returns *nbyte* .
- If O_NONBLOCK and O_NDELAY are set, write( ) does not block the process. If a write( ) request for PIPE_BUF or fewer bytes succeeds completely write( ) returns *nbyte* . Otherwise, if O_NONBLOCK is set, it returns –1 and sets errno to EAGAIN or if O_NDELAY is set, it returns 0 .

A `write( )` request for greater than `{PIPE_BUF}` bytes transfers what it can and returns the number of bytes written or it transfers no data and, if `O_NONBLOCK` is set, returns `–1` with `errno` set to `EAGAIN` or if `O_NDELAY` is set, it returns `0` . Finally, if a request is greater than `PIPE_BUF` bytes and all data previously written to the pipe has been read, `write( )` transfers at least `PIPE_BUF` bytes.

When attempting to write to a file descriptor (other than a pipe, a FIFO , a socket, or a STREAM) that supports nonblocking writes and cannot accept the data immediately:

- If `O_NONBLOCK` and `O_NDELAY` are clear, `write( )` blocks until the data can be accepted.
- If `O_NONBLOCK` or `O_NDELAY` is set, `write( )` does not block the process. If some data can be written without blocking the process, `write( )` writes what it can and returns the number of bytes written. Otherwise, if `O_NONBLOCK` is set, it returns `–1` and sets `errno` to `EAGAIN` or if `O_NDELAY` is set, it returns `0` .

Upon successful completion, where *nbyte* is greater than 0, `write( )` will mark for update the `st_ctime` and `st_mtime` fields of the file, and if the file is a regular file, the `S_ISUID` and `S_ISGID` bits of the file mode may be cleared.

For STREAMS files (see `intro`(3) and `streamio`(7I)), the operation of `write( )` is determined by the values of the minimum and maximum *nbyte* range (“packet size”) accepted by the STREAM. These values are contained in the topmost STREAM module, and 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 non-zero, `write( )` fails and sets `errno` to `ERANGE` . Writing a zero-length buffer (*nbyte* is zero) to a STREAMS device sends a zero length message with zero returned. However, writing a zero-length buffer to a pipe or FIFO sends no message and zero is returned. The user program may issue the `I_SWROPT` `ioctl`(2) to enable zero-length messages to be sent across the pipe or FIFO (see `streamio`(7I))..

When writing to a STREAM, data messages are created with a priority band of zero. When writing to a socket or to a STREAM that is not a pipe or a FIFO :

- If `O_NDELAY` and `O_NONBLOCK` are not set, and the STREAM cannot accept data (the STREAM write queue is full due to internal flow control conditions), `write( )` blocks until data can be accepted.
- If `O_NDELAY` or `O_NONBLOCK` is set and the STREAM cannot accept data, `write( )` returns `–1` and sets `errno` to `EAGAIN` .

- If O_NDELAY or O_NONBLOCK is set and part of the buffer has already been written when a condition occurs in which the STREAM cannot accept additional data, write( ) terminates and returns the number of bytes written.

The write( ) and writev( ) functions will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of errno does not reflect the result of write( ) or writev( ) but reflects the prior error.

pwrite( ) The pwrite( ) function performs the same action as write( ) , except that it writes into a given position without changing the file pointer. The first three arguments to pwrite( ) are the same as write( ) with the addition of a fourth argument *offset* for the desired position inside the file.

writev( ) The writev( ) function performs the same action as write( ) , but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: *iov* [0], *iov* [1], …, *iov* [*iovcnt* −1]. The *iovcnt* buffer is valid if greater than 0 and less than or equal to IOV_MAX . See intro(2) for a definition of IOV_MAX .

The iovec structure contains the following members:

```
caddr_t  iov_base;
int      iov_len;
```

Each iovec entry specifies the base address and length of an area in memory from which data should be written. The writev( ) function always writes all data from an area before proceeding to the next.

If *fildes* refers to a regular file and all of the iov_len members in the array pointed to by *iov* are 0, writev( ) will return 0 and have no other effect. For other file types, the behavior is unspecified.

If the sum of the iov_len values is greater than SSIZE_MAX , the operation fails and no data is transferred.

writel( ), pwritel( ), and writevl( ) 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* has this restriction:

- If the descriptor refers to a file or a FIFO , then the sensitivity label portion of *label_p* is ignored.

In all other respects, the writel( ) , pwritel( ) , and writevl( ) interfaces are analogous to the write( ) , pwrite( ) , and writev( ) interfaces.

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.

RETURN VALUES

Upon successful completion, write( ) returns the number of bytes actually written to the file associated with *fildes* . This number is never greater than *nbyte* . Otherwise, –1 is returned, the file-pointer remains unchanged, and errno is set to indicate the error.

Upon successful completion, writev( ) returns the number of bytes actually written. Otherwise, it returns –1 , the file-pointer remains unchanged, and errno is set to indicate an error.

ERRORS

The write( ) , pwrite( ) , writev( ) writel( ) , pwritel( ) , and writevl( ) functions will fail if:

EAGAIN — Mandatory file / record locking is set, O_NDELAY or O_NONBLOCK is set, and there is a blocking record lock; total amount of system memory available when reading using raw I / O is temporarily insufficient; an attempt is made to write to a STREAM that can not accept data with the O_NDELAY or O_NONBLOCK flag set; or a write to a pipe or FIFO of PIPE_BUF bytes or less is requested and less than *nbytes* of free space is available.

EBADF — The *fildes* argument is not a valid file descriptor open for writing.

EDEADLK — The write was going to go to sleep and cause a deadlock situation to occur.

EDQUOT — The user's quota of disk blocks on the file system containing the file has been exhausted.

EFAULT — The *buf* argument points to an illegal address.

EFBIG — An attempt is made to write a file that exceeds the process's file size limit or the maximum file size (see getrlimit(2) and ulimit(2)).

The file is a regular file, *nbyte* is greater than 0, and the starting position is greater than or equal to the offset

maximum established in the file description associated with *fildes* .

EINTR — A signal was caught during the write operation and no data was transferred.

EIO — The process is in the background and is attempting to write to its controlling terminal whose TOSTOP flag is set, or the process is neither ignoring nor blocking SIGTTOU signals and the process group of the process is orphaned.

ENOLCK — Enforced record locking was enabled and LOCK_MAX regions are already locked in the system, or the system record lock table was full and the write could not go to sleep until the blocking record lock was removed.

ENOLINK — The *fildes* argument is on a remote machine and the link to that machine is no longer active.

ENOSPC — During a write to an ordinary file, there is no free space left on the device.

ENOSR — An attempt is made to write to a STREAMS with insufficient STREAMS memory resources available in the system.

ENXIO — A hangup occurred on the STREAM being written to.

EPIPE — An attempt is made to write to a pipe or a FIFO that is not open for reading by any process, or that has only one end open (or to a file descriptor created by socket(3SOCKET) , using type SOCK_STREAM that is no longer connected to a peer endpoint). A SIGPIPE signal will also be sent to the process. The process dies unless special provisions were taken to catch or ignore the signal.

ERANGE — The transfer request size was outside the range supported by the STREAMS file associated with *fildes* .

The pwrite( ) function fails and the file pointer remains unchanged if:

ESPIPE — The *fildes* argument is associated with a pipe or FIFO .

The writev( ) function will fail if:

EINVAL — The sum of the iov_len values in the *iov* array would overflow an ssize_t .

The write( ) and writev( ) functions may fail if:

EINVAL — The STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device.

ENXIO A hangup occurred on the STREAM being written to.

A write to a STREAMS file may fail if an error message has been received at the STREAM head. In this case, errno is set to the value included in the error message.

The writev( ) function may fail if:

EINVAL The *iovcnt* argument was less than or equal to 0 or greater than IOV_MAX ; one of the iov_len values in the *iov* array was negative; or the sum of the iov_len values in the *iov* array overflowed an int .

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.

USAGE

The pwrite( ) function has a transitional interface for 64-bit file offsets. See lf64(5) .

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	write( ) is Async-Signal-Safe

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.

SEE ALSO

Trusted Solaris 8 Reference Manual

chmod(2) , creat(2) , fcntl(2) , getrlimit(2) , lseek(2) , open(2) , ulimit(2) , intro(3) , send(3SOCKET)

SunOS 5.8 Reference Manual

dup(2) , ioctl(2) , pipe(2) , socket(3SOCKET) , attributes(5) , lf64(5) , streamio(7I)

NAME

write, pwrite, writev, writel, pwritel, writevl – Write on a file

SYNOPSIS

```
#include <unistd.h>
ssize_t write(int fildes, const void * buf, size_t nbyte);

ssize_t pwrite(int fildes, const void * buf, size_t nbyte, off_t offset);

#include <sys/uio.h>
ssize_t writev(int fildes, const struct iovec * iov, int iovcnt);

#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);
```

DESCRIPTION

The `write()` function attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor, *fildes* .

If *nbyte* is `0` , `write()` will return `0` and have no other results if the file is a regular file; otherwise, the results are unspecified.

On a regular file or other file capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file offset associated with *fildes* . Before successful return from `write()` , the file offset is incremented by the number of bytes actually written. On a regular file, if this incremented file offset is greater than the length of the file, the length of the file will be set to this file offset.

If the `O_SYNC` flag of the file status flags is set and *fildes* refers to a regular file, a successful `write()` does not return until the data is delivered to the underlying hardware.

If *fildes* refers to a socket, `write()` is equivalent to `send`(3SOCKET) with no flags set.

On a file not capable of seeking, writing always takes place starting at the current position. The value of a file offset associated with such a device is undefined.

If the `O_APPEND` flag of the file status flags is set, the file offset will be set to the end of the file prior to each write and no intervening file modification operation will occur between changing the file offset and the write operation.

For regular files, no data transfer will occur past the offset maximum established in the open file description with *fildes* .

A `write()` to a regular file is blocked if mandatory file / record locking is set (see `chmod`(2)), and there is a record lock owned by another process on the segment of the file to be written:

- If O_NDELAY or O_NONBLOCK is set, write( ) returns –1 and sets errno to EAGAIN .
- If O_NDELAY and O_NONBLOCK are clear, write( ) sleeps until all blocking locks are removed or the write( ) is terminated by a signal.

If a write( ) requests that more bytes be written than there is room for—for example, if the write would exceed the process file size limit (see getrlimit(2) and ulimit(2)), the system file size limit, or the free space on the device—only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write( ) of 512-bytes returns 20 . The next write( ) of a non-zero number of bytes gives a failure return (except as noted for pipes and FIFO below).

If write( ) is interrupted by a signal before it writes any data, it will return –1 with errno set to EINTR .

If write( ) is interrupted by a signal after it successfully writes some data, it will return the number of bytes written.

If the value of *nbyte* is greater than SSIZE_MAX , the result is implementation-dependent.

After a write( ) to a regular file has successfully returned:

- Any successful read(2) from each byte position in the file that was modified by that write will return the data specified by the write( ) for that position until such byte positions are again modified.
- Any subsequent successful write( ) to the same byte position in the file will overwrite that file data.

Write requests to a pipe or FIFO are handled the same as a regular file with the following exceptions:

- There is no file offset associated with a pipe, hence each write request appends to the end of the pipe.
- Write requests of {PIPE_BUF} bytes or less are guaranteed not to be interleaved with data from other processes doing writes on the same pipe. Writes of greater than {PIPE_BUF} bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the O_NONBLOCK or O_NDELAY flags are set.
- If O_NONBLOCK and O_NDELAY are clear, a write request may cause the process to block, but on normal completion it returns *nbyte* .
- If O_NONBLOCK and O_NDELAY are set, write( ) does not block the process. If a write( ) request for PIPE_BUF or fewer bytes succeeds completely write( ) returns *nbyte* . Otherwise, if O_NONBLOCK is set, it returns –1 and sets errno to EAGAIN or if O_NDELAY is set, it returns 0 .

A `write( )` request for greater than `{PIPE_BUF}` bytes transfers what it can and returns the number of bytes written or it transfers no data and, if `O_NONBLOCK` is set, returns `–1` with `errno` set to `EAGAIN` or if `O_NDELAY` is set, it returns `0` . Finally, if a request is greater than `PIPE_BUF` bytes and all data previously written to the pipe has been read, `write( )` transfers at least `PIPE_BUF` bytes.

When attempting to write to a file descriptor (other than a pipe, a FIFO , a socket, or a STREAM) that supports nonblocking writes and cannot accept the data immediately:

- If `O_NONBLOCK` and `O_NDELAY` are clear, `write( )` blocks until the data can be accepted.
- If `O_NONBLOCK` or `O_NDELAY` is set, `write( )` does not block the process. If some data can be written without blocking the process, `write( )` writes what it can and returns the number of bytes written. Otherwise, if `O_NONBLOCK` is set, it returns `–1` and sets `errno` to `EAGAIN` or if `O_NDELAY` is set, it returns `0` .

Upon successful completion, where *nbyte* is greater than 0, `write( )` will mark for update the `st_ctime` and `st_mtime` fields of the file, and if the file is a regular file, the `S_ISUID` and `S_ISGID` bits of the file mode may be cleared.

For STREAMS files (see `intro`(3) and `streamio`(7I)), the operation of `write( )` is determined by the values of the minimum and maximum *nbyte* range (“packet size”) accepted by the STREAM. These values are contained in the topmost STREAM module, and 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 non-zero, `write( )` fails and sets `errno` to `ERANGE` . Writing a zero-length buffer (*nbyte* is zero) to a STREAMS device sends a zero length message with zero returned. However, writing a zero-length buffer to a pipe or FIFO sends no message and zero is returned. The user program may issue the `I_SWROPT` `ioctl`(2) to enable zero-length messages to be sent across the pipe or FIFO (see `streamio`(7I))..

When writing to a STREAM, data messages are created with a priority band of zero. When writing to a socket or to a STREAM that is not a pipe or a FIFO :

- If `O_NDELAY` and `O_NONBLOCK` are not set, and the STREAM cannot accept data (the STREAM write queue is full due to internal flow control conditions), `write( )` blocks until data can be accepted.
- If `O_NDELAY` or `O_NONBLOCK` is set and the STREAM cannot accept data, `write( )` returns `–1` and sets `errno` to `EAGAIN` .

- If O_NDELAY or O_NONBLOCK is set and part of the buffer has already been written when a condition occurs in which the STREAM cannot accept additional data, write( ) terminates and returns the number of bytes written.

The write( ) and writev( ) functions will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of errno does not reflect the result of write( ) or writev( ) but reflects the prior error.

pwrite( ) The pwrite( ) function performs the same action as write( ) , except that it writes into a given position without changing the file pointer. The first three arguments to pwrite( ) are the same as write( ) with the addition of a fourth argument *offset* for the desired position inside the file.

writev( ) The writev( ) function performs the same action as write( ) , but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: *iov* [0], *iov* [1], …, *iov* [*iovcnt* −1]. The *iovcnt* buffer is valid if greater than 0 and less than or equal to IOV_MAX . See intro(2) for a definition of IOV_MAX .

The iovec structure contains the following members:

```
caddr_t  iov_base;
int      iov_len;
```

Each iovec entry specifies the base address and length of an area in memory from which data should be written. The writev( ) function always writes all data from an area before proceeding to the next.

If *fildes* refers to a regular file and all of the iov_len members in the array pointed to by *iov* are 0, writev( ) will return 0 and have no other effect. For other file types, the behavior is unspecified.

If the sum of the iov_len values is greater than SSIZE_MAX , the operation fails and no data is transferred.

writel( ), pwritel( ), and writevl( ) 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* has this restriction:

- If the descriptor refers to a file or a FIFO , then the sensitivity label portion of *label_p* is ignored.

In all other respects, the writel( ) , pwritel( ) , and writevl( ) interfaces are analogous to the write( ) , pwrite( ) , and writev( ) interfaces.

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.

RETURN VALUES

Upon successful completion, write( ) returns the number of bytes actually written to the file associated with *fildes* . This number is never greater than *nbyte* . Otherwise, –1 is returned, the file-pointer remains unchanged, and errno is set to indicate the error.

Upon successful completion, writev( ) returns the number of bytes actually written. Otherwise, it returns –1 , the file-pointer remains unchanged, and errno is set to indicate an error.

ERRORS

The write( ) , pwrite( ) , writev( ) writel( ) , pwritel( ) , and writevl( ) functions will fail if:

EAGAIN — Mandatory file / record locking is set, O_NDELAY or O_NONBLOCK is set, and there is a blocking record lock; total amount of system memory available when reading using raw I / O is temporarily insufficient; an attempt is made to write to a STREAM that can not accept data with the O_NDELAY or O_NONBLOCK flag set; or a write to a pipe or FIFO of PIPE_BUF bytes or less is requested and less than *nbytes* of free space is available.

EBADF — The *fildes* argument is not a valid file descriptor open for writing.

EDEADLK — The write was going to go to sleep and cause a deadlock situation to occur.

EDQUOT — The user's quota of disk blocks on the file system containing the file has been exhausted.

EFAULT — The *buf* argument points to an illegal address.

EFBIG — An attempt is made to write a file that exceeds the process's file size limit or the maximum file size (see getrlimit(2) and ulimit(2)).

The file is a regular file, *nbyte* is greater than 0, and the starting position is greater than or equal to the offset

maximum established in the file description associated with *fildes* .

EINTR — A signal was caught during the write operation and no data was transferred.

EIO — The process is in the background and is attempting to write to its controlling terminal whose TOSTOP flag is set, or the process is neither ignoring nor blocking SIGTTOU signals and the process group of the process is orphaned.

ENOLCK — Enforced record locking was enabled and LOCK_MAX regions are already locked in the system, or the system record lock table was full and the write could not go to sleep until the blocking record lock was removed.

ENOLINK — The *fildes* argument is on a remote machine and the link to that machine is no longer active.

ENOSPC — During a write to an ordinary file, there is no free space left on the device.

ENOSR — An attempt is made to write to a STREAMS with insufficient STREAMS memory resources available in the system.

ENXIO — A hangup occurred on the STREAM being written to.

EPIPE — An attempt is made to write to a pipe or a FIFO that is not open for reading by any process, or that has only one end open (or to a file descriptor created by socket(3SOCKET) , using type SOCK_STREAM that is no longer connected to a peer endpoint). A SIGPIPE signal will also be sent to the process. The process dies unless special provisions were taken to catch or ignore the signal.

ERANGE — The transfer request size was outside the range supported by the STREAMS file associated with *fildes* .

The pwrite( ) function fails and the file pointer remains unchanged if:

ESPIPE — The *fildes* argument is associated with a pipe or FIFO .

The writev( ) function will fail if:

EINVAL — The sum of the iov_len values in the *iov* array would overflow an ssize_t .

The write( ) and writev( ) functions may fail if:

EINVAL — The STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device.

ENXIO A hangup occurred on the STREAM being written to.

A write to a STREAMS file may fail if an error message has been received at the STREAM head. In this case, errno is set to the value included in the error message.

The writev( ) function may fail if:

EINVAL The *iovcnt* argument was less than or equal to 0 or greater than IOV_MAX ; one of the iov_len values in the *iov* array was negative; or the sum of the iov_len values in the *iov* array overflowed an int .

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.

USAGE

The pwrite( ) function has a transitional interface for 64-bit file offsets. See lf64(5) .

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	write( ) is Async-Signal-Safe

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.

SEE ALSO

Trusted Solaris 8 Reference Manual

chmod(2) , creat(2) , fcntl(2) , getrlimit(2) , lseek(2) , open(2) , ulimit(2) , intro(3) , send(3SOCKET)

SunOS 5.8 Reference Manual

dup(2) , ioctl(2) , pipe(2) , socket(3SOCKET) , attributes(5) , lf64(5) , streamio(7I)

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);
```

DESCRIPTION

The `read( )` function attempts to read *nbyte* bytes from the file associated with the open file descriptor, *fildes* , into the buffer pointed to by *buf* .

If *nbyte* is 0, `read( )` will return `0` and have no other results.

On files that support seeking (for example, a regular file), the `read( )` starts at a position in the file given by the file offset associated with *fildes* . The file offset is incremented by the number of bytes actually read.

Files that do not support seeking (for example, terminals) always read from the current position. The value of a file offset associated with such a file is undefined.

If *fildes* refers to a socket, `read( )` is equivalent to `recv`(3SOCKET) with no flags set.

No data transfer will occur past the current end-of-file. If the starting position is at or after the end-of-file, `0` will be returned. If the file refers to a device special file, the result of subsequent `read( )` requests is implementation-dependent.

If the value of *nbyte* is greater than `SSIZE_MAX` , the result is implementation-dependent.

When attempting to read from a regular file with mandatory file / record locking set (see `chmod`(2)), and there is a write lock owned by another process on the segment of the file to be read:

- If `O_NDELAY` or `O_NONBLOCK` is set, `read( )` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read( )` sleeps until the blocking record lock is removed.

When attempting to read from an empty pipe (or FIFO):

- If no process has the pipe open for writing, `read( )` returns `0` to indicate end-of-file.
- If some process has the pipe open for writing and `O_NDELAY` is set, `read( )` returns `0` .
- If some process has the pipe open for writing and `O_NONBLOCK` is set, `read( )` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read( )` blocks until data is written to the pipe or the pipe is closed by all processes that had opened the pipe for writing.

When attempting to read a file associated with a terminal that has no data currently available:

- If `O_NDELAY` is set, `read( )` returns `0` .
- If `O_NONBLOCK` is set, `read( )` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read( )` blocks until data become available.

When attempting to read a file associated with a socket or a stream that is not a pipe, a FIFO , or a terminal, and the file has no data currently available:

- If `O_NDELAY` or `O_NONBLOCK` is set, `read( )` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read( )` blocks until data becomes available.

The `read( )` function reads data previously written to a file. If any portion of a regular file prior to the end-of-file has not been written, `read( )` returns bytes with value `0` . For example, `lseek`(2) allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return bytes with value 0 until data is written into the gap.

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with *fildes* .

Upon successful completion, where *nbyte* is greater than 0, `read( )` will mark for update the `st_atime` field of the file, and return the number of bytes read. This number will never be greater than *nbyte* . The value returned may be less than *nbyte* if the number of bytes left in the file is less than *nbyte* , if the `read( )` request was interrupted by a signal, or if the file is a pipe or FIFO or special file and has fewer than *nbyte* bytes immediately available for reading. For example, a `read( )` from a file associated with a terminal may return one typed line of data.

If a `read( )` is interrupted by a signal before it reads any data, it will return `–1` with `errno` set to `EINTR` .

If a `read( )` is interrupted by a signal after it has successfully read some data, it will return the number of bytes read.

A `read( )` or `readv( )` from a STREAMS (see `intro`(2)) file can read data in three different modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default is byte-stream mode. This can be changed using the `I_SRDOPT` `ioctl`(2) request, and can be tested with the `I_GRDOPT` `ioctl( )` . In byte-stream mode, `read( )` retrieves data from the STREAM until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, `read( )` retrieves data until as many bytes as were requested are transferred, or until a message boundary is reached. If `read( )` does not retrieve all the data in a message, the remaining data is left on the STREAM, and can be retrieved by the next `read( )` call. Message-discard mode also retrieves data until as many bytes as were requested are transferred, or a message boundary is reached. However, unread data remaining in a message after the `read( )` returns is discarded, and is not available for a subsequent `read( )` , `readv( )` or `getmsg`(2) call.

How `read( )` handles zero-byte STREAMS messages is determined by the current read mode setting. In byte-stream mode, `read( )` accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The `read( )` function then returns the number of bytes read, and places the zero-byte message back on the STREAM to be retrieved by the next `read( )` , `readv( )` or `getmsg`(2) . In message-nondiscard mode or message-discard mode, a zero-byte message returns `0` and the message is removed from the STREAM. When a zero-byte message is read as the first message on a STREAM, the message is removed from the STREAM and `0` is returned, regardless of the read mode.

A `read( )` from a STREAMS file returns the data in the message at the front of the STREAM head read queue, regardless of the priority band of the message.

By default, STREAMS are in control-normal mode, in which a `read( )` from a STREAMS file can only process messages that contain a data part but do not contain a control part. The `read( )` fails if a message containing a control part is encountered at the STREAM head. This default action can be changed by placing the STREAM in either control-data mode or control-discard mode with the `I_SRDOPT` `ioctl( )` command. In control-data mode, `read( )` converts any control part to data and passes it to the application before passing any data part originally present in the same message. In control-discard mode, `read( )` discards message control parts but returns to the process any data part in the message.

`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 filesystem object.

In all other respects, the `readl( )` , `preadl( )` , and `readvl( )` interfaces are analogous to the `read( )` , `pread( )` , and `readv( )` interfaces.

In the SunOS 5.7 operating system, `read( )` normally allows a process to read the contents of directories on some local file systems. This functionality is not supported in the Trusted Solaris operating environment. 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 filesystem object. There is no privilege to override this restriction.

In addition, `read( )` and `readv( )` will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of `errno` does not reflect the result of `read( )` or `readv( )` but reflects the prior error. If a hangup occurs on the STREAM being read, `read( )` continues to operate normally until the STREAM head read queue is empty. Thereafter, it returns `0` .

`readv( )`

The `readv( )` function is equivalent to `read( )` , but places the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov* `0` , *iov* `1` , ..., *iov* [*iovcnt* –1]. The *iovcnt* argument is valid if greater than 0 and less than or equal to `IOV_MAX` .

The `iovec` structure contains the following members:

```
caddr_t   iov_base;
int       iov_len;
```

Each `iovec` entry specifies the base address and length of an area in memory where data should be placed. The `readv( )` function always fills an area completely before proceeding to the next.

Upon successful completion, `readv( )` marks for update the `st_atime` field of the file.

`pread( )`

The `pread( )` function performs the same action as `read( )` , except that it reads from a given position in the file without changing the file pointer. The first three arguments to `pread( )` are the same as `read( )` with the addition of a fourth argument *offset* for the desired position inside the file. `pread( )` will read up to the maximum offset value that can be represented in an `off_t` for regular files. An attempt to perform a `pread( )` on a file that is incapable of seeking results in an error.

RETURN VALUES

Upon successful completion, `read()` and `readv()` return a non-negative integer indicating the number of bytes actually read. Otherwise, the functions return `–1` and set `errno` to indicate the error.

ERRORS

`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; or no message is waiting to be read on a stream and `O_NDELAY` or `O_NONBLOCK` was set.

`EBADF` The *fildes* argument is not a valid file descriptor open for reading.

`EBADMSG` Message waiting to be read on a stream is not a data message.

`EDEADLK` The read was going to go to sleep and cause a deadlock to occur.

`EFAULT` The *buf* argument points to an illegal address.

`EINTR` A signal was caught during the read operation and no data was transferred.

`EINVAL` An attempt was made to read from a stream linked to a multiplexor.

`EIO` A physical I/O error has occurred, or the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the `SIGTTIN` signal or the process group of the process is orphaned.

`EISDIR` The *fildes* argument refers to a directory on a file system type that does not support read operations on directories.

`ENOLCK` The system record lock table was full, so the `read()` or `readv()` could not go to sleep until the blocking record lock was removed.

`ENOLINK` The *fildes* argument is on a remote machine and the link to that machine is no longer active.

ENXIO The device associated with *fildes* is a block special or character special file and the value of the file pointer is out of range.

The `read()` and `readv()` functions will fail if:

EOVERFLOW The file is a regular file, *nbyte* is greater than 0, the starting position is before the end-of-file, and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fildes* .

The `readv()` function may fail if:

EFAULT The *iov* argument points outside the allocated address space.

EINVAL The *iovcnt* argument was less than or equal to `0` , or greater than or equal to `IOV_MAX` . (See `intro`(2) for a definition of `IOV_MAX`).

EINVAL The sum of the `iov_len` values in the *iov* array overflowed an int.

The `pread()` function will fail and the file pointer remain unchanged if:

EFAULT *label_p* points to an illegal address.

USAGE

The `pread()` function has a transitional interface for 64-bit file offsets. See `lf64`(5) .

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	`read()` is Async-Signal-Safe

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 filesystem object. There is no privilege to override this restriction.

For conduits, a sensitivity label is associated with each byte of data.

SEE ALSO

Trusted Solaris 8 Reference Manual

`intro`(2) , `chmod`(2) , `creat`(2) , `fcntl`(2) , `open`(2)

SunOS 5.8 Reference Manual

`dup`(2) , `getmsg`(2) , `ioctl`(2) , `pipe`(2) , `streamio`(7I) , `termio`(7I)

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);
```

DESCRIPTION

The `read( )` function attempts to read *nbyte* bytes from the file associated with the open file descriptor, *fildes* , into the buffer pointed to by *buf* .

If *nbyte* is 0, `read( )` will return `0` and have no other results.

On files that support seeking (for example, a regular file), the `read( )` starts at a position in the file given by the file offset associated with *fildes* . The file offset is incremented by the number of bytes actually read.

Files that do not support seeking (for example, terminals) always read from the current position. The value of a file offset associated with such a file is undefined.

If *fildes* refers to a socket, `read( )` is equivalent to `recv`(3SOCKET) with no flags set.

No data transfer will occur past the current end-of-file. If the starting position is at or after the end-of-file, `0` will be returned. If the file refers to a device special file, the result of subsequent `read( )` requests is implementation-dependent.

If the value of *nbyte* is greater than `SSIZE_MAX` , the result is implementation-dependent.

When attempting to read from a regular file with mandatory file / record locking set (see `chmod`(2)), and there is a write lock owned by another process on the segment of the file to be read:

- If `O_NDELAY` or `O_NONBLOCK` is set, `read( )` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read( )` sleeps until the blocking record lock is removed.

When attempting to read from an empty pipe (or FIFO):

- If no process has the pipe open for writing, `read()` returns `0` to indicate end-of-file.
- If some process has the pipe open for writing and `O_NDELAY` is set, `read()` returns `0` .
- If some process has the pipe open for writing and `O_NONBLOCK` is set, `read()` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read()` blocks until data is written to the pipe or the pipe is closed by all processes that had opened the pipe for writing.

When attempting to read a file associated with a terminal that has no data currently available:

- If `O_NDELAY` is set, `read()` returns `0` .
- If `O_NONBLOCK` is set, `read()` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read()` blocks until data become available.

When attempting to read a file associated with a socket or a stream that is not a pipe, a FIFO , or a terminal, and the file has no data currently available:

- If `O_NDELAY` or `O_NONBLOCK` is set, `read()` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read()` blocks until data becomes available.

The `read()` function reads data previously written to a file. If any portion of a regular file prior to the end-of-file has not been written, `read()` returns bytes with value `0` . For example, `lseek`(2) allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return bytes with value 0 until data is written into the gap.

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with *fildes* .

Upon successful completion, where *nbyte* is greater than 0, `read()` will mark for update the `st_atime` field of the file, and return the number of bytes read. This number will never be greater than *nbyte* . The value returned may be less than *nbyte* if the number of bytes left in the file is less than *nbyte* , if the `read()` request was interrupted by a signal, or if the file is a pipe or FIFO or special file and has fewer than *nbyte* bytes immediately available for reading. For example, a `read()` from a file associated with a terminal may return one typed line of data.

If a `read()` is interrupted by a signal before it reads any data, it will return `–1` with `errno` set to `EINTR` .

If a `read( )` is interrupted by a signal after it has successfully read some data, it will return the number of bytes read.

A `read( )` or `readv( )` from a STREAMS (see `intro`(2)) file can read data in three different modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default is byte-stream mode. This can be changed using the `I_SRDOPT` `ioctl`(2) request, and can be tested with the `I_GRDOPT` `ioctl( )` . In byte-stream mode, `read( )` retrieves data from the STREAM until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, `read( )` retrieves data until as many bytes as were requested are transferred, or until a message boundary is reached. If `read( )` does not retrieve all the data in a message, the remaining data is left on the STREAM, and can be retrieved by the next `read( )` call. Message-discard mode also retrieves data until as many bytes as were requested are transferred, or a message boundary is reached. However, unread data remaining in a message after the `read( )` returns is discarded, and is not available for a subsequent `read( )` , `readv( )` or `getmsg`(2) call.

How `read( )` handles zero-byte STREAMS messages is determined by the current read mode setting. In byte-stream mode, `read( )` accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The `read( )` function then returns the number of bytes read, and places the zero-byte message back on the STREAM to be retrieved by the next `read( )` , `readv( )` or `getmsg`(2) . In message-nondiscard mode or message-discard mode, a zero-byte message returns `0` and the message is removed from the STREAM. When a zero-byte message is read as the first message on a STREAM, the message is removed from the STREAM and `0` is returned, regardless of the read mode.

A `read( )` from a STREAMS file returns the data in the message at the front of the STREAM head read queue, regardless of the priority band of the message.

By default, STREAMS are in control-normal mode, in which a `read( )` from a STREAMS file can only process messages that contain a data part but do not contain a control part. The `read( )` fails if a message containing a control part is encountered at the STREAM head. This default action can be changed by placing the STREAM in either control-data mode or control-discard mode with the `I_SRDOPT` `ioctl( )` command. In control-data mode, `read( )` converts any control part to data and passes it to the application before passing any data part originally present in the same message. In control-discard mode, `read( )` discards message control parts but returns to the process any data part in the message.

`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 filesystem object.

In all other respects, the `readl( )` , `preadl( )` , and `readvl( )` interfaces are analogous to the `read( )` , `pread( )` , and `readv( )` interfaces.

In the SunOS 5.7 operating system, `read( )` normally allows a process to read the contents of directories on some local file systems. This functionality is not supported in the Trusted Solaris operating environment. 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 filesystem object. There is no privilege to override this restriction.

In addition, `read( )` and `readv( )` will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of `errno` does not reflect the result of `read( )` or `readv( )` but reflects the prior error. If a hangup occurs on the STREAM being read, `read( )` continues to operate normally until the STREAM head read queue is empty. Thereafter, it returns `0` .

`readv( )` The `readv( )` function is equivalent to `read( )` , but places the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov* `0` , *iov* `1` , ..., *iov* [*iovcnt* –1]. The *iovcnt* argument is valid if greater than 0 and less than or equal to `IOV_MAX` .

The `iovec` structure contains the following members:

```
caddr_t   iov_base;
int       iov_len;
```

Each `iovec` entry specifies the base address and length of an area in memory where data should be placed. The `readv( )` function always fills an area completely before proceeding to the next.

Upon successful completion, `readv( )` marks for update the `st_atime` field of the file.

`pread( )` The `pread( )` function performs the same action as `read( )` , except that it reads from a given position in the file without changing the file pointer. The first three arguments to `pread( )` are the same as `read( )` with the addition of a fourth argument *offset* for the desired position inside the file. `pread( )` will read up to the maximum offset value that can be represented in an `off_t` for regular files. An attempt to perform a `pread( )` on a file that is incapable of seeking results in an error.

RETURN VALUES

Upon successful completion, `read()` and `readv()` return a non-negative integer indicating the number of bytes actually read. Otherwise, the functions return `–1` and set `errno` to indicate the error.

ERRORS

`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; or no message is waiting to be read on a stream and `O_NDELAY` or `O_NONBLOCK` was set.
`EBADF`	The *fildes* argument is not a valid file descriptor open for reading.
`EBADMSG`	Message waiting to be read on a stream is not a data message.
`EDEADLK`	The read was going to go to sleep and cause a deadlock to occur.
`EFAULT`	The *buf* argument points to an illegal address.
`EINTR`	A signal was caught during the read operation and no data was transferred.
`EINVAL`	An attempt was made to read from a stream linked to a multiplexor.
`EIO`	A physical I/O error has occurred, or the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the `SIGTTIN` signal or the process group of the process is orphaned.
`EISDIR`	The *fildes* argument refers to a directory on a file system type that does not support read operations on directories.
`ENOLCK`	The system record lock table was full, so the `read()` or `readv()` could not go to sleep until the blocking record lock was removed.
`ENOLINK`	The *fildes* argument is on a remote machine and the link to that machine is no longer active.

ENXIO The device associated with *fildes* is a block special or character special file and the value of the file pointer is out of range.

The `read( )` and `readv( )` functions will fail if:

EOVERFLOW The file is a regular file, *nbyte* is greater than 0, the starting position is before the end-of-file, and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fildes* .

The `readv( )` function may fail if:

EFAULT The *iov* argument points outside the allocated address space.

EINVAL The *iovcnt* argument was less than or equal to `0` , or greater than or equal to `IOV_MAX` . (See `intro`(2) for a definition of `IOV_MAX`).

EINVAL The sum of the `iov_len` values in the *iov* array overflowed an int.

The `pread( )` function will fail and the file pointer remain unchanged if:

EFAULT *label_p* points to an illegal address.

USAGE

The `pread( )` function has a transitional interface for 64-bit file offsets. See `lf64`(5) .

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	`read( )` is Async-Signal-Safe

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 filesystem object. There is no privilege to override this restriction.

For conduits, a sensitivity label is associated with each byte of data.

SEE ALSO

Trusted Solaris 8 Reference Manual

`intro`(2) , `chmod`(2) , `creat`(2) , `fcntl`(2) , `open`(2)

SunOS 5.8 Reference Manual

`dup`(2) , `getmsg`(2) , `ioctl`(2) , `pipe`(2) , `streamio`(7I) , `termio`(7I)

NAME

readlink – Read the contents of a symbolic link

SYNOPSIS

```
#include <unistd.h>
int readlink(const char *path, char *buf, size_t bufsiz);
```

DESCRIPTION

The `readlink( )` function places the contents of the symbolic link referred to by *path* in the buffer *buf* which has size *bufsiz*. If the number of bytes in the symbolic link is less than *bufsiz*, the contents of the remainder of *buf* are unspecified.

RETURN VALUES

Upon successful completion, `readlink( )` returns the count of bytes placed in the buffer. Otherwise, it returns `–1`, leaves the buffer unchanged, and sets `errno` to indicate the error.

ERRORS

The `readlink( )` function will fail if:

`EACCES`	Search permission is denied for a component of the path prefix of *path*. 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 *path* argument names a file that is not a symbolic link.
`EIO`	An I/O error occurred while reading from the file system.
`ENOENT`	A component of *path* does not name an existing file or *path* is an empty string.
`ELOOP`	Too many symbolic links were encountered in resolving *path*.
`ENAMETOOLONG`	The length of *path* exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.
`ENOTDIR`	A component of the path prefix is not a directory.
`ENOSYS`	The file system does not support symbolic links.

The `readlink( )` function may fail if:

`EACCES`	Read permission is denied for the directory.

ENAMETOOLONG	Path name resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.

USAGE

Portable applications should not assume that the returned contents of the symbolic link are null-terminated.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

Trusted Solaris 8 Reference Manual

stat(2), symlink(2)

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);
```

DESCRIPTION

The `read( )` function attempts to read *nbyte* bytes from the file associated with the open file descriptor, *fildes* , into the buffer pointed to by *buf* .

If *nbyte* is 0, `read( )` will return `0` and have no other results.

On files that support seeking (for example, a regular file), the `read( )` starts at a position in the file given by the file offset associated with *fildes* . The file offset is incremented by the number of bytes actually read.

Files that do not support seeking (for example, terminals) always read from the current position. The value of a file offset associated with such a file is undefined.

If *fildes* refers to a socket, `read( )` is equivalent to `recv`(3SOCKET) with no flags set.

No data transfer will occur past the current end-of-file. If the starting position is at or after the end-of-file, `0` will be returned. If the file refers to a device special file, the result of subsequent `read( )` requests is implementation-dependent.

If the value of *nbyte* is greater than `SSIZE_MAX` , the result is implementation-dependent.

When attempting to read from a regular file with mandatory file / record locking set (see `chmod`(2)), and there is a write lock owned by another process on the segment of the file to be read:

- If `O_NDELAY` or `O_NONBLOCK` is set, `read( )` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read( )` sleeps until the blocking record lock is removed.

When attempting to read from an empty pipe (or FIFO):

- If no process has the pipe open for writing, `read( )` returns `0` to indicate end-of-file.
- If some process has the pipe open for writing and `O_NDELAY` is set, `read( )` returns `0` .
- If some process has the pipe open for writing and `O_NONBLOCK` is set, `read( )` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read( )` blocks until data is written to the pipe or the pipe is closed by all processes that had opened the pipe for writing.

When attempting to read a file associated with a terminal that has no data currently available:

- If `O_NDELAY` is set, `read( )` returns `0` .
- If `O_NONBLOCK` is set, `read( )` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read( )` blocks until data become available.

When attempting to read a file associated with a socket or a stream that is not a pipe, a FIFO , or a terminal, and the file has no data currently available:

- If `O_NDELAY` or `O_NONBLOCK` is set, `read( )` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read( )` blocks until data becomes available.

The `read( )` function reads data previously written to a file. If any portion of a regular file prior to the end-of-file has not been written, `read( )` returns bytes with value `0` . For example, `lseek`(2) allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return bytes with value 0 until data is written into the gap.

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with *fildes* .

Upon successful completion, where *nbyte* is greater than 0, `read( )` will mark for update the `st_atime` field of the file, and return the number of bytes read. This number will never be greater than *nbyte* . The value returned may be less than *nbyte* if the number of bytes left in the file is less than *nbyte* , if the `read( )` request was interrupted by a signal, or if the file is a pipe or FIFO or special file and has fewer than *nbyte* bytes immediately available for reading. For example, a `read( )` from a file associated with a terminal may return one typed line of data.

If a `read( )` is interrupted by a signal before it reads any data, it will return `–1` with `errno` set to `EINTR` .

If a `read( )` is interrupted by a signal after it has successfully read some data, it will return the number of bytes read.

A `read( )` or `readv( )` from a STREAMS (see `intro`(2)) file can read data in three different modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default is byte-stream mode. This can be changed using the `I_SRDOPT` `ioctl`(2) request, and can be tested with the `I_GRDOPT` `ioctl( )` . In byte-stream mode, `read( )` retrieves data from the STREAM until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, `read( )` retrieves data until as many bytes as were requested are transferred, or until a message boundary is reached. If `read( )` does not retrieve all the data in a message, the remaining data is left on the STREAM, and can be retrieved by the next `read( )` call. Message-discard mode also retrieves data until as many bytes as were requested are transferred, or a message boundary is reached. However, unread data remaining in a message after the `read( )` returns is discarded, and is not available for a subsequent `read( )` , `readv( )` or `getmsg`(2) call.

How `read( )` handles zero-byte STREAMS messages is determined by the current read mode setting. In byte-stream mode, `read( )` accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The `read( )` function then returns the number of bytes read, and places the zero-byte message back on the STREAM to be retrieved by the next `read( )` , `readv( )` or `getmsg`(2) . In message-nondiscard mode or message-discard mode, a zero-byte message returns `0` and the message is removed from the STREAM. When a zero-byte message is read as the first message on a STREAM, the message is removed from the STREAM and `0` is returned, regardless of the read mode.

A `read( )` from a STREAMS file returns the data in the message at the front of the STREAM head read queue, regardless of the priority band of the message.

By default, STREAMS are in control-normal mode, in which a `read( )` from a STREAMS file can only process messages that contain a data part but do not contain a control part. The `read( )` fails if a message containing a control part is encountered at the STREAM head. This default action can be changed by placing the STREAM in either control-data mode or control-discard mode with the `I_SRDOPT` `ioctl( )` command. In control-data mode, `read( )` converts any control part to data and passes it to the application before passing any data part originally present in the same message. In control-discard mode, `read( )` discards message control parts but returns to the process any data part in the message.

`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 filesystem object.

In all other respects, the `readl( )` , `preadl( )` , and `readvl( )` interfaces are analogous to the `read( )` , `pread( )` , and `readv( )` interfaces.

In the SunOS 5.7 operating system, `read( )` normally allows a process to read the contents of directories on some local file systems. This functionality is not supported in the Trusted Solaris operating environment. 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 filesystem object. There is no privilege to override this restriction.

In addition, `read( )` and `readv( )` will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of `errno` does not reflect the result of `read( )` or `readv( )` but reflects the prior error. If a hangup occurs on the STREAM being read, `read( )` continues to operate normally until the STREAM head read queue is empty. Thereafter, it returns `0` .

`readv( )` The `readv( )` function is equivalent to `read( )` , but places the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov* `0` , *iov* `1` , ..., *iov* [*iovcnt* –1]. The *iovcnt* argument is valid if greater than 0 and less than or equal to `IOV_MAX` .

The `iovec` structure contains the following members:

```
caddr_t   iov_base;
int       iov_len;
```

Each `iovec` entry specifies the base address and length of an area in memory where data should be placed. The `readv( )` function always fills an area completely before proceeding to the next.

Upon successful completion, `readv( )` marks for update the `st_atime` field of the file.

`pread( )` The `pread( )` function performs the same action as `read( )` , except that it reads from a given position in the file without changing the file pointer. The first three arguments to `pread( )` are the same as `read( )` with the addition of a fourth argument *offset* for the desired position inside the file. `pread( )` will read up to the maximum offset value that can be represented in an `off_t` for regular files. An attempt to perform a `pread( )` on a file that is incapable of seeking results in an error.

RETURN VALUES

Upon successful completion, `read()` and `readv()` return a non-negative integer indicating the number of bytes actually read. Otherwise, the functions return `–1` and set `errno` to indicate the error.

ERRORS

`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; or no message is waiting to be read on a stream and `O_NDELAY` or `O_NONBLOCK` was set.

`EBADF` The *fildes* argument is not a valid file descriptor open for reading.

`EBADMSG` Message waiting to be read on a stream is not a data message.

`EDEADLK` The read was going to go to sleep and cause a deadlock to occur.

`EFAULT` The *buf* argument points to an illegal address.

`EINTR` A signal was caught during the read operation and no data was transferred.

`EINVAL` An attempt was made to read from a stream linked to a multiplexor.

`EIO` A physical I/O error has occurred, or the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the `SIGTTIN` signal or the process group of the process is orphaned.

`EISDIR` The *fildes* argument refers to a directory on a file system type that does not support read operations on directories.

`ENOLCK` The system record lock table was full, so the `read()` or `readv()` could not go to sleep until the blocking record lock was removed.

`ENOLINK` The *fildes* argument is on a remote machine and the link to that machine is no longer active.

ENXIO The device associated with *fildes* is a block special or character special file and the value of the file pointer is out of range.

The `read( )` and `readv( )` functions will fail if:

EOVERFLOW The file is a regular file, *nbyte* is greater than 0, the starting position is before the end-of-file, and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fildes* .

The `readv( )` function may fail if:

EFAULT The *iov* argument points outside the allocated address space.

EINVAL The *iovcnt* argument was less than or equal to `0` , or greater than or equal to `IOV_MAX` . (See `intro`(2) for a definition of `IOV_MAX`).

EINVAL The sum of the `iov_len` values in the *iov* array overflowed an int.

The `pread( )` function will fail and the file pointer remain unchanged if:

EFAULT *label_p* points to an illegal address.

USAGE

The `pread( )` function has a transitional interface for 64-bit file offsets. See `lf64`(5) .

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	`read( )` is Async-Signal-Safe

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 filesystem object. There is no privilege to override this restriction.

For conduits, a sensitivity label is associated with each byte of data.

SEE ALSO

Trusted Solaris 8 Reference Manual

`intro`(2) , `chmod`(2) , `creat`(2) , `fcntl`(2) , `open`(2)

SunOS 5.8 Reference Manual

`dup`(2) , `getmsg`(2) , `ioctl`(2) , `pipe`(2) , `streamio`(7I) , `termio`(7I)

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);
```

DESCRIPTION

The `read( )` function attempts to read *nbyte* bytes from the file associated with the open file descriptor, *fildes* , into the buffer pointed to by *buf* .

If *nbyte* is 0, `read( )` will return `0` and have no other results.

On files that support seeking (for example, a regular file), the `read( )` starts at a position in the file given by the file offset associated with *fildes* . The file offset is incremented by the number of bytes actually read.

Files that do not support seeking (for example, terminals) always read from the current position. The value of a file offset associated with such a file is undefined.

If *fildes* refers to a socket, `read( )` is equivalent to `recv`(3SOCKET) with no flags set.

No data transfer will occur past the current end-of-file. If the starting position is at or after the end-of-file, `0` will be returned. If the file refers to a device special file, the result of subsequent `read( )` requests is implementation-dependent.

If the value of *nbyte* is greater than `SSIZE_MAX` , the result is implementation-dependent.

When attempting to read from a regular file with mandatory file / record locking set (see `chmod`(2)), and there is a write lock owned by another process on the segment of the file to be read:

- If `O_NDELAY` or `O_NONBLOCK` is set, `read( )` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read( )` sleeps until the blocking record lock is removed.

When attempting to read from an empty pipe (or FIFO):

- If no process has the pipe open for writing, `read()` returns `0` to indicate end-of-file.
- If some process has the pipe open for writing and `O_NDELAY` is set, `read()` returns `0` .
- If some process has the pipe open for writing and `O_NONBLOCK` is set, `read()` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read()` blocks until data is written to the pipe or the pipe is closed by all processes that had opened the pipe for writing.

When attempting to read a file associated with a terminal that has no data currently available:

- If `O_NDELAY` is set, `read()` returns `0` .
- If `O_NONBLOCK` is set, `read()` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read()` blocks until data become available.

When attempting to read a file associated with a socket or a stream that is not a pipe, a FIFO , or a terminal, and the file has no data currently available:

- If `O_NDELAY` or `O_NONBLOCK` is set, `read()` returns `–1` and sets `errno` to `EAGAIN` .
- If `O_NDELAY` and `O_NONBLOCK` are clear, `read()` blocks until data becomes available.

The `read()` function reads data previously written to a file. If any portion of a regular file prior to the end-of-file has not been written, `read()` returns bytes with value `0` . For example, `lseek`(2) allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return bytes with value 0 until data is written into the gap.

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with *fildes* .

Upon successful completion, where *nbyte* is greater than 0, `read()` will mark for update the `st_atime` field of the file, and return the number of bytes read. This number will never be greater than *nbyte* . The value returned may be less than *nbyte* if the number of bytes left in the file is less than *nbyte* , if the `read()` request was interrupted by a signal, or if the file is a pipe or FIFO or special file and has fewer than *nbyte* bytes immediately available for reading. For example, a `read()` from a file associated with a terminal may return one typed line of data.

If a `read()` is interrupted by a signal before it reads any data, it will return `–1` with `errno` set to `EINTR` .

If a `read( )` is interrupted by a signal after it has successfully read some data, it will return the number of bytes read.

A `read( )` or `readv( )` from a STREAMS (see `intro`(2)) file can read data in three different modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default is byte-stream mode. This can be changed using the `I_SRDOPT` `ioctl`(2) request, and can be tested with the `I_GRDOPT` `ioctl( )` . In byte-stream mode, `read( )` retrieves data from the STREAM until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, `read( )` retrieves data until as many bytes as were requested are transferred, or until a message boundary is reached. If `read( )` does not retrieve all the data in a message, the remaining data is left on the STREAM, and can be retrieved by the next `read( )` call. Message-discard mode also retrieves data until as many bytes as were requested are transferred, or a message boundary is reached. However, unread data remaining in a message after the `read( )` returns is discarded, and is not available for a subsequent `read( )` , `readv( )` or `getmsg`(2) call.

How `read( )` handles zero-byte STREAMS messages is determined by the current read mode setting. In byte-stream mode, `read( )` accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The `read( )` function then returns the number of bytes read, and places the zero-byte message back on the STREAM to be retrieved by the next `read( )` , `readv( )` or `getmsg`(2) . In message-nondiscard mode or message-discard mode, a zero-byte message returns `0` and the message is removed from the STREAM. When a zero-byte message is read as the first message on a STREAM, the message is removed from the STREAM and `0` is returned, regardless of the read mode.

A `read( )` from a STREAMS file returns the data in the message at the front of the STREAM head read queue, regardless of the priority band of the message.

By default, STREAMS are in control-normal mode, in which a `read( )` from a STREAMS file can only process messages that contain a data part but do not contain a control part. The `read( )` fails if a message containing a control part is encountered at the STREAM head. This default action can be changed by placing the STREAM in either control-data mode or control-discard mode with the `I_SRDOPT` `ioctl( )` command. In control-data mode, `read( )` converts any control part to data and passes it to the application before passing any data part originally present in the same message. In control-discard mode, `read( )` discards message control parts but returns to the process any data part in the message.

`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 filesystem object.

In all other respects, the `readl( )` , `preadl( )` , and `readvl( )` interfaces are analogous to the `read( )` , `pread( )` , and `readv( )` interfaces.

In the SunOS 5.7 operating system, `read( )` normally allows a process to read the contents of directories on some local file systems. This functionality is not supported in the Trusted Solaris operating environment. 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 filesystem object. There is no privilege to override this restriction.

In addition, `read( )` and `readv( )` will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of `errno` does not reflect the result of `read( )` or `readv( )` but reflects the prior error. If a hangup occurs on the STREAM being read, `read( )` continues to operate normally until the STREAM head read queue is empty. Thereafter, it returns `0` .

`readv( )` The `readv( )` function is equivalent to `read( )` , but places the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov* `0` , *iov* `1` , ..., *iov* [*iovcnt* –1]. The *iovcnt* argument is valid if greater than 0 and less than or equal to `IOV_MAX` .

The `iovec` structure contains the following members:

```
caddr_t   iov_base;
int       iov_len;
```

Each `iovec` entry specifies the base address and length of an area in memory where data should be placed. The `readv( )` function always fills an area completely before proceeding to the next.

Upon successful completion, `readv( )` marks for update the `st_atime` field of the file.

`pread( )` The `pread( )` function performs the same action as `read( )` , except that it reads from a given position in the file without changing the file pointer. The first three arguments to `pread( )` are the same as `read( )` with the addition of a fourth argument *offset* for the desired position inside the file. `pread( )` will read up to the maximum offset value that can be represented in an `off_t` for regular files. An attempt to perform a `pread( )` on a file that is incapable of seeking results in an error.

RETURN VALUES

Upon successful completion, `read()` and `readv()` return a non-negative integer indicating the number of bytes actually read. Otherwise, the functions return `–1` and set `errno` to indicate the error.

ERRORS

`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; or no message is waiting to be read on a stream and `O_NDELAY` or `O_NONBLOCK` was set.

`EBADF` The *fildes* argument is not a valid file descriptor open for reading.

`EBADMSG` Message waiting to be read on a stream is not a data message.

`EDEADLK` The read was going to go to sleep and cause a deadlock to occur.

`EFAULT` The *buf* argument points to an illegal address.

`EINTR` A signal was caught during the read operation and no data was transferred.

`EINVAL` An attempt was made to read from a stream linked to a multiplexor.

`EIO` A physical I/O error has occurred, or the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the `SIGTTIN` signal or the process group of the process is orphaned.

`EISDIR` The *fildes* argument refers to a directory on a file system type that does not support read operations on directories.

`ENOLCK` The system record lock table was full, so the `read()` or `readv()` could not go to sleep until the blocking record lock was removed.

`ENOLINK` The *fildes* argument is on a remote machine and the link to that machine is no longer active.

ENXIO The device associated with *fildes* is a block special or character special file and the value of the file pointer is out of range.

The `read( )` and `readv( )` functions will fail if:

EOVERFLOW The file is a regular file, *nbyte* is greater than 0, the starting position is before the end-of-file, and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fildes* .

The `readv( )` function may fail if:

EFAULT The *iov* argument points outside the allocated address space.

EINVAL The *iovcnt* argument was less than or equal to `0` , or greater than or equal to `IOV_MAX` . (See `intro`(2) for a definition of `IOV_MAX`).

EINVAL The sum of the `iov_len` values in the *iov* array overflowed an int.

The `pread( )` function will fail and the file pointer remain unchanged if:

EFAULT *label_p* points to an illegal address.

USAGE

The `pread( )` function has a transitional interface for 64-bit file offsets. See `lf64`(5) .

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	`read( )` is Async-Signal-Safe

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 filesystem object. There is no privilege to override this restriction.

For conduits, a sensitivity label is associated with each byte of data.

SEE ALSO

Trusted Solaris 8 Reference Manual

`intro`(2) , `chmod`(2) , `creat`(2) , `fcntl`(2) , `open`(2)

SunOS 5.8 Reference Manual

`dup`(2) , `getmsg`(2) , `ioctl`(2) , `pipe`(2) , `streamio`(7I) , `termio`(7I)

NAME

rename – Change the name of a file

SYNOPSIS

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

DESCRIPTION

The `rename()` function changes the name of a file. The *old* argument points to the pathname of the file to be renamed. The *new* argument points to the new pathname of the file.

If *old* and *new* both refer to the same existing file, the `rename()` function returns successfully and performs no other action.

If *old* points to the pathname of a file that is not a directory, *new* must not point to the pathname of a directory. If the link named by *new* exists, it will be removed and *old* will be renamed to *new*. In this case, a link named *new* must remain visible to other processes throughout the renaming operation and will refer to either the file referred to by *new* or the file referred to as *old* before the operation began.

If *old* points to the pathname of a directory, *new* must not point to the pathname of a file that is not a directory. If the directory named by *new* exists, it will be removed and *old* will be renamed to *new*. In this case, a link named *new* will exist throughout the renaming operation and will refer to either the file referred to by *new* or the file referred to as *old* before the operation began. Thus, if *new* names an existing directory, it must be an empty directory.

The *new* pathname must not contain a path prefix that names *old*. Write access permission is required for both the directory containing *old* and the directory containing *new*. If *old* points to the pathname of a directory, write access permission is required for the directory named by *old*, and, if it exists, the directory named by *new*.

If the directory containing *old* has the sticky bit set, at least one of the following conditions listed below must be true:

- The user must own *old*
- The user must own the directory containing *old*
- *old* must be writable by the user
- The user must be a privileged user

If *new* exists, and the directory containing *new* is writable and has the sticky bit set, at least one of the following conditions must be true:

- the user must own *new*
- the user must own the directory containing *new*
- *new* must be writable by the user
- the user must be a privileged user

If the link named by *new* exists, the file's link count becomes zero when it is removed, and no process has the file open, then the space occupied by the file will be freed and the file will no longer be accessible. If one or more processes have the file open when the last link is removed, the link will be removed before `rename()` returns, but the removal of the file contents will be postponed until all references to the file have been closed.

Upon successful completion, the `rename()` function will mark for update the `st_ctime` and `st_mtime` fields of the parent directory of each file.

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.

RETURN VALUES

`rename()` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `rename()` function will fail if:

`EACCES` A component of either path prefix denies search permission; one of the directories containing *old* and *new* denies write permissions; or write permission is denied by a directory pointed to by *old* or *new*. 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` The *new* argument is a directory and the mount point for a mounted file system.

`EDQUOT` The directory where the new name entry is being placed cannot be extended because the user's quota of disk blocks on that file system has been exhausted.

`EEXIST` The link named by *new* is a directory containing entries other than '`.`' (the directory itself) and '`..`' (the parent directory).

`EINVAL` The *new* argument directory pathname contains a path prefix that names the *old* directory.

EISDIR	The *new* argument points to a directory but *old* points to a file that is not a directory.
ELOOP	Too many symbolic links were encountered in translating the pathname.
ENAMETOOLONG	The length of *old* or *new* exceeds PATH_MAX, or a pathname component is longer than NAME_MAX while _POSIX_NO_TRUNC is in effect.
EMLINK	The file named by *old* is a directory, and the link count of the parent directory of *new* would exceed LINK_MAX.
ENOENT	The link named by *old* does not exist, or either *old* or *new* points to an empty string.
ENOSPC	The directory that would contain *new* cannot be extended.
ENOTDIR	A component of either path prefix is not a directory, or *old* names a directory and *new* names a nondirectory file.
EROFS	The requested operation requires writing in a directory on a read-only file system.
EXDEV	The links named by *old* and *new* are on different file systems.
EIO	An I/O error occurred while making or updating a directory entry.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

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
Trusted Solaris 8 Reference Manual

chmod(2), link(2), unlink(2)

SunOS 5.8 Reference Manual

attributes(5)

WARNINGS

The system can deadlock if there is a loop in the file system graph. Such a loop can occur if there is an entry in directory a, a/name1, that is a hard link to directory b, and an entry in directory b, b/name2, that is a hard link to directory a. When such a loop exists and two separate processes attempt to rename a/name1 to b/name2 and b/name2 to a/name1, the system may deadlock attempting to lock both directories for modification. The solution is to use symbolic links instead of hard links for directories.

NAME

rmdir – Remove a directory

SYNOPSIS

```
#include <unistd.h>
int rmdir(const char *path);
```

DESCRIPTION

The `rmdir( )` function removes the directory named by the path name pointed to by *path*. The directory must not have any entries other than "`.`" and "`..`".

If the directory's link count becomes zero and no process has the directory open, the space occupied by the directory is freed and the directory is no longer accessible. If one or more processes have the directory open when the last link is removed, the "`.`" and "`..`" entries, if present, are removed before `rmdir( )` returns and no new entries may be created in the directory, but the directory is not removed until all references to the directory have been closed.

Upon successful completion `rmdir( )` marks for update the `st_ctime` and `st_mtime` fields of the parent directory. A multilevel directory can be removed only when all its contained single-level directories are empty.

RETURN VALUES

`rmdir( )` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `rmdir( )` function will fail if:

`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 those for "." and "..".
EFAULT	The *path* argument points to an illegal address.
EINVAL	The directory to be removed is the current directory, or the final component of *path* is ".".
EIO	An I/O error occurred while accessing the file system.
ELOOP	Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of the *path* argument exceeds PATH_MAX, or the length of a *path* component exceeds NAME_MAX while _POSIX_NO_TRUNC is in effect.
ENOENT	The named directory does not exist or is the null pathname.
ENOLINK	The *path* argument points to a remote machine, and the connection to that machine is no longer active.
ENOTDIR	A component of the path prefix is not a directory.
EROFS	The directory entry to be removed is part of a read-only file system.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

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.

SEE ALSO

Trusted Solaris 8 Reference Manual

mkdir(1), rm(1), mkdir(2)

SunOS 5.8 Reference Manual

attributes(5)

NAME

secconf – Get security configuration information

SYNOPSIS

cc [*flags*…] *file*… –ltsol

#include <tsol/secconf.h>
long **secconf**(int *name*);

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	_TSOL_CLEAN_WINDOWS
Variable name	tsol_clean_windows
	Force cleaning of unused register windows before return from system call (SPARC architecture only).
int name	_TSOL_FLUSH_BUFFERS
Variable name	tsol_flush_buffers
	Force flushing of file data blocks before inode updates.
int name	_TSOL_HIDE_UPGRADED_NAMES
Variable name	tsol_hide_upgraded_names
	Hide upgrade directory entries.
int name	_TSOL_PRIVS_DEBUG
Variable name	tsol_privs_debug
	Enables privilege debugging mode.

RETURN VALUES

secconf( ) returns:

0	On success.
–1	When *name* is an invalid value. Also sets errno to indicate the error.
	When *name* is not defined on the system. The value of errno will not be set.

ERRORS

The function will return the following errors:

EINVAL	The parameter *name* is unknown.

FILES

system(4)	System configuration information file.

SEE ALSO

Trusted Solaris 8 Reference Manual

pathconf(2)

SunOS 5.8 Reference Manual

`sysconf`(3C)

Trusted Solaris Information Label Changes

Information labels (ILs) are not supported in Trusted Solaris 7 and later releases. Trusted Solaris software interprets any ILs on communications and files from systems running earlier releases as `ADMIN_LOW`.

Objects still have CMW labels, and CMW labels still include the IL component: `IL[SL]`; however, the IL component is fixed at `ADMIN_LOW`.

As a result, Trusted Solaris 7 and later releases have the following characteristics:

- ILs do not display in window labels; SLs (Sensitivity Labels) display alone within brackets.
- ILs do not float.
- Setting an IL on an object has no effect.
- Getting an object's IL will always return `ADMIN_LOW`.
- Although certain utilities, library functions, and system calls can manipulate IL strings, the resulting ILs cannot be set on any objects.
- Sensitivity labels, not information labels, display on printer banners.

NAME

semctl – Semaphore control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

DESCRIPTION

The `semctl( )` function provides a variety of semaphore control operations as specified by *cmd*. The fourth argument is optional, depending upon the operation requested. If required, it is of type `union semun`, which must be explicitly declared by the application program.

```
union semun {
        int             val;
        struct semid_ds *buf;
        ushort_t        *array;
} arg ;
```

The permission required for a semaphore operation is given as {*token*}, where *token* is the type of permission needed. The types of permission are interpreted as follows:

```
00400   READ by user
00200   ALTER by user
00040   READ by group
00020   ALTER by group
00004   READ by others
00002   ALTER by others
```

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`(2) for more information. The following semaphore operations as specified by *cmd* are executed with respect to the semaphore specified by *semid* and *semnum*.

`GETVAL` Return the value of semval (see `intro`(2)). {READ}

SETVAL Set the value of `semval` to *arg.val*. {ALTER} When this command is successfully executed, the `semadj` value corresponding to the specified semaphore in all processes is cleared.

GETPID Return the value of `(int) sempid`. {READ}

GETNCNT Return the value of `semncnt`. {READ}

GETZCNT Return the value of `semzcnt`. {READ}

The following operations return and set, respectively, every `semval` in the set of semaphores.

GETALL Place `semval`s into array pointed to by *arg.*`array`. {READ}

SETALL Set `semval`s according to the array pointed to by *arg.*`array`. {ALTER}. When this cmd is successfully executed, the `semadj` values corresponding to each specified semaphore in all processes are cleared.

The following operations are also available.

IPC_STAT Place the current value of each member of the data structure associated with *semid* into the structure pointed to by *arg.*`buf`. The contents of this structure are defined in `intro`(2). {READ}

IPC_SET Set the value of the following members of the data structure associated with *semid* to the corresponding value found in the structure pointed to by *arg.*`buf`:

```
sem_perm.uid
sem_perm.gid
sem_perm.mode /* access permission bits only */
```

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* as follows:

GETVAL	the value of semval
GETPID	the value of (int) sempid
GETNCNT	the value of semncnt
GETZCNT	the value of semzcnt

All other successful completions return 0; otherwise, −1 is returned and errno is set to indicate the error.

ERRORS

The semctl( ) function will fail if:

EACCES	Operation permission is denied to the calling process (see intro(2)), and the process lacks the appropriate privilege override(s) in its set of effective privileges.
EINVAL	The *semid* argument is not a valid semaphore identifier; the *semnum* argument is less than 0 or greater than sem_nsems −1; or the *cmd* argument is not a valid command or is IPC_SET and sem_perm.uid or sem_perm.gid is not valid.
EPERM	*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	The *cmd* argument is IPC_STAT and *uid* or *gid* is too large to be stored in the structure pointed to by *arg.buf*.
ERANGE	The *cmd* argument is SETVAL or SETALL and the value to which semval is to be set is greater than the system imposed maximum.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

Trusted Solaris 8 Reference Manual

`ipcs`(1), `intro`(2), `semget`(2), `semop`(2)

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 [flags...] 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 the following is true:

- *key* is equal to `IPC_PRIVATE` .
- *key* does not already have a semaphore identifier associated with it, and (*semflg* &`IPC_CREAT`) is true.

On creation, the data structure associated with the new semaphore identifier is initialized as follows:

- `sem_perm.cuid` , `sem_perm.uid` , `sem_perm.cgid` , and `sem_perm.gid` are set equal to the effective user ID and effective group ID , respectively, of the calling process.
- The access permission bits of `sem_perm.mode` are set equal to the access permission bits of *semflg* .
- `sem_nsems` is set equal to the value of *nsems* .
- `sem_otime` is set equal to 0 and `sem_ctime` is set equal to the current time.

RETURN VALUES

Upon successful completion, a non-negative integer representing a semaphore identifier is returned. Otherwise, –1 is returned and `errno` is set to indicate the error.

ERRORS

The `semget( )` function will fail if:

`EACCES` A semaphore-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 both (*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.

`EINVAL` The *nsems* argument is either less than or equal to 0 or greater than the system-imposed limit; or a semaphore identifier exists for *key* , but the number of semaphores in the set associated with it is less than *nsems* and *nsems* is not equal to 0.

`ENOENT` A semaphore identifier does not exist for *key* and (*semflg* `&IPC_CREAT`) is false.

`ENOSPC` A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores or semaphore identifiers system-wide would be exceeded.

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

Trusted Solaris 8 Reference Manual

`ipcs`(1) , `ipcrm`(1) , `intro`(2) , `semctl`(2) , `semop`(2)

SunOS 5.8 Reference Manual

`stdio`(3C)

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 [*flags*…] *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 the following is true:

- *key* is equal to `IPC_PRIVATE` .
- *key* does not already have a semaphore identifier associated with it, and (*semflg* `&IPC_CREAT`) is true.

On creation, the data structure associated with the new semaphore identifier is initialized as follows:

- `sem_perm.cuid` , `sem_perm.uid` , `sem_perm.cgid` , and `sem_perm.gid` are set equal to the effective user ID and effective group ID , respectively, of the calling process.
- The access permission bits of `sem_perm.mode` are set equal to the access permission bits of *semflg* .
- `sem_nsems` is set equal to the value of *nsems* .
- `sem_otime` is set equal to 0 and `sem_ctime` is set equal to the current time.

RETURN VALUES

Upon successful completion, a non-negative integer representing a semaphore identifier is returned. Otherwise, –1 is returned and `errno` is set to indicate the error.

ERRORS

The `semget( )` function will fail if:

`EACCES` A semaphore-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 both (*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.

`EINVAL` The *nsems* argument is either less than or equal to 0 or greater than the system-imposed limit; or a semaphore identifier exists for *key* , but the number of semaphores in the set associated with it is less than *nsems* and *nsems* is not equal to 0.

`ENOENT` A semaphore identifier does not exist for *key* and (*semflg* `&IPC_CREAT`) is false.

`ENOSPC` A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores or semaphore identifiers system-wide would be exceeded.

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

Trusted Solaris 8 Reference Manual

`ipcs`(1) , `ipcrm`(1) , `intro`(2) , `semctl`(2) , `semop`(2)

SunOS 5.8 Reference Manual

`stdio`(3C)

NAME

semop – Semaphore operations

SYNOPSIS

```
#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>
int semop( int semid, struct sembuf *sops, size_t nsops);
```

DESCRIPTION

The `semop( )` function is used to perform atomically an array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by *semid*. The *sops* argument is a pointer to the array of semaphore-operation structures. The *nsops* argument is the number of such structures in the array.

Each `sembuf` structure contains the following members:

```
short sem_num;   /* semaphore number */
short sem_op;     /* semaphore operation */
short sem_flg;   /* operation flags */
```

Each semaphore operation specified by `sem_op( )` is performed on the corresponding semaphore specified by *semid* and *sem_num*. The permission required for a semaphore operation is given as {*token*}, where *token* is the type of permission needed. The types of permission are interpreted as follows:

```
00400    READ by user
00200    ALTER by user
00040    READ by group
00020    ALTER by group
00004    READ by others
00002    ALTER by others
```

See the *Semaphore Operation Permissions* section of `intro`(2) for more information.

The `sem_op` member specifies one of three semaphore operations:

1. The `sem_op` member is a negative integer; {ALTER}

   - If `semval` (see `intro`(2)) is greater than or equal to the absolute value of `sem_op`, the absolute value of `sem_op` is subtracted from `semval`. Also, if (*sem_flg*&`SEM_UNDO`) is true, the absolute value of `sem_op` is added to the calling process's `semadj` value (see `exit`(2)) for the specified semaphore.

- If `semval` is less than the absolute value of `sem_op` and (*sem_flg*`&IPC_NOWAIT`) is true, `semop( )` returns immediately.
- If `semval` is less than the absolute value of `sem_op` and (*sem_flg*`&IPC_NOWAIT`) is false, `semop( )` increments the `semncnt` associated with the specified semaphore and suspends execution of the calling process until one of the following conditions occur:
  - The value of `semval` becomes greater than or equal to the absolute value of `sem_op`. When this occurs, the value of `semncnt` associated with the specified semaphore is decremented, the absolute value of `sem_op` is subtracted from `semval` and, if (*sem_flg*`&SEM_UNDO`) is true, the absolute value of `sem_op` is added to the calling process's `semadj` value for the specified semaphore.
  - The *semid* for which the calling process is awaiting action is removed from the system (see `semctl`(2)). When this occurs, `errno` is set to `EIDRM` and `−1` is returned.
  - The calling process receives a signal that is to be caught. When this occurs, the value of `semncnt` associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in `signal`(3C).

2. The `sem_op` member is a positive integer; {ALTER}

   The value of `sem_op` is added to `semval` and, if (*sem_flg*`&SEM_UNDO`) is true, the value of `sem_op` is subtracted from the calling process's `semadj` value for the specified semaphore.

3. The `sem_op` member is 0; {READ}
   - If `semval` is 0, `semop( )` returns immediately.
   - If `semval` is not equal to 0 and (*sem_flg*`&IPC_NOWAIT`) is true, `semop( )` returns immediately.
   - If `semval` is not equal to 0 and (*sem_flg*`&IPC_NOWAIT`) is false, `semop( )` increments the `semzcnt` associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:
     - The value of `semval` becomes 0, at which time the value of `semzcnt` associated with the specified semaphore is decremented.
     - The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, `errno` is set to `EIDRM` and `−1` is returned.

- The calling process receives a signal that is to be caught. When this occurs, the value of `semzcnt` associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in `signal`(3C).

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`.

Upon successful completion, the value of `sempid` for each semaphore specified in the array pointed to by *sops* is set to the process ID of the calling process.

RETURN VALUES

`semop( )` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `semop( )` function will fail if:

`E2BIG` The *nsops* argument is greater than the system-imposed maximum.

`EACCES` Operation permission is denied to the calling process (see `intro`(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` The *sops* argument points to an illegal address.

`EFBIG` The value of `sem_num` is less than 0 or greater than or equal to the number of semaphores in the set associated with *semid*.

`EIDRM` A *semid* was removed from the system.

`EINTR` A signal was received.

`EINVAL` The *semid* argument is not a valid semaphore identifier, or the number of individual semaphores for which the calling process requests a `SEM_UNDO` would exceed the limit.

ENOSPC	The limit on the number of individual processes requesting an SEM_UNDO would be exceeded.
ERANGE	An operation would cause a semval or a semadj value to overflow the system-imposed limit.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

Trusted Solaris 8 Reference Manual

ipcs(1), intro(2), exec(2), fork(2), semctl(2), semget(2)

SunOS 5.8 Reference Manual

signal(3C)

NAME

getaudit, setaudit, getaudit_addr, setaudit_addr – 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*);

int **getaudit_addr**(struct auditinfo_addr * *info*, int *length*);

int **setaudit_addr**(struct auditinfo_addr * *info*, int *length*);

DESCRIPTION

`getaudit( )` gets the audit ID , the preselection mask, the terminal ID , and the audit session ID of the current process.

Note that `getaudit( )` may fail and return an `E2BIG` errno if the address field in the terminal ID is larger than 32 bits. In this case, `getaudit_addr( )` should be used.

`setaudit( )` sets the audit ID , the preselection mask, the terminal ID , and the audit session ID for the current process.

The `getaudit_addr( )` function returns a variable length `auditinfo_addr` structure that contains the audit ID , the preselection mask, the terminal ID , and the audit session ID for the current process. The terminal ID contains a size field that indicates the size of the network address.

The `setaudit_addr( )` function sets the audit ID , the preselection mask, the terminal ID , and the audit session ID for the current process. The values are taken from the variable length struture `auditinfo_addr.` The terminal ID contains a size field that indicates the size of the network address.

The `info` structure used to pass the process audit information contains the following members:

```
au_id_t     ai_auid;        /* audit user ID */
au_mask_t   ai_mask;        /* preselection mask */
au_tid_t    ai_termid;      /* terminal ID */
au_asid_t   ai_asid;        /* audit session ID */
```

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

`getaudit( )` and `setaudit( )` return:

`0` On success.

`–1` On failure, and set `errno` to indicate the error.

ERRORS

The `getaudit( )` and `setaudit( )` functions will fail if:

`EFAULT` The *info* parameter points outside the process's allocated address space.

`EPERM` The process did not have the appropriate privilege.

USAGE

Only processes with the appropriate privileges may successfully execute these calls.

SUMMARY OF TRUSTED SOLARIS CHANGES

This functionality is active only if auditing is enabled. By default, auditing is enabled in the Trusted Solaris environment. See *Trusted Solaris Audit Administration* for more information.

As explained in `DESCRIPTION` , privileges are needed to run this command successfully.

SEE ALSO

Trusted Solaris 8 Reference Manual

`audit`(2)

NAME

getaudit, setaudit, getaudit_addr, setaudit_addr – 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*);

int **getaudit_addr**(struct auditinfo_addr * *info*, int *length*);

int **setaudit_addr**(struct auditinfo_addr * *info*, int *length*);

DESCRIPTION

`getaudit( )` gets the audit ID , the preselection mask, the terminal ID , and the audit session ID of the current process.

Note that `getaudit( )` may fail and return an `E2BIG` errno if the address field in the terminal ID is larger than 32 bits. In this case, `getaudit_addr( )` should be used.

`setaudit( )` sets the audit ID , the preselection mask, the terminal ID , and the audit session ID for the current process.

The `getaudit_addr( )` function returns a variable length `auditinfo_addr` structure that contains the audit ID , the preselection mask, the terminal ID , and the audit session ID for the current process. The terminal ID contains a size field that indicates the size of the network address.

The `setaudit_addr( )` function sets the audit ID , the preselection mask, the terminal ID , and the audit session ID for the current process. The values are taken from the variable length struture `auditinfo_addr.` The terminal ID contains a size field that indicates the size of the network address.

The `info` structure used to pass the process audit information contains the following members:

```
au_id_t     ai_auid;        /* audit user ID */
au_mask_t   ai_mask;        /* preselection mask */
au_tid_t    ai_termid;      /* terminal ID */
au_asid_t   ai_asid;        /* audit session ID */
```

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

`getaudit( )` and `setaudit( )` return:

`0` On success.

`–1` On failure, and set `errno` to indicate the error.

ERRORS

The `getaudit( )` and `setaudit( )` functions will fail if:

`EFAULT` The *info* parameter points outside the process's allocated address space.

`EPERM` The process did not have the appropriate privilege.

USAGE

Only processes with the appropriate privileges may successfully execute these calls.

SUMMARY OF TRUSTED SOLARIS CHANGES

This functionality is active only if auditing is enabled. By default, auditing is enabled in the Trusted Solaris environment. See *Trusted Solaris Audit Administration* for more information.

As explained in `DESCRIPTION` , privileges are needed to run this command successfully.

SEE ALSO

Trusted Solaris 8 Reference Manual

`audit`(2)

NAME

getauid, setauid – Get and set user audit identity

SYNOPSIS

cc [*flags*…] *file* … –lbsm –lsocket –lnsl –lintl [*library*…]

```
#include <sys/param.h>

#include <bsm/audit.h>
int getauid(au_id_t * auid);

int setauid(au_id_t * auid);
```

DESCRIPTION

The `getauid( )` function returns the audit user ID for the current process. This value is initially set at login time and inherited by all child processes. This value does not change when the real / effective user ID s change, so it can be used to identify the logged-in user even when running a setuid program. The audit user ID governs audit decisions for a process.

The `setauid( )` function sets the audit user ID for the current process.

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 successful completion, the `getauid( )` function returns the audit user ID of the current process on success. Otherwise, it returns `–1` and sets `errno` to indicate the error.

Upon successful completion the `setauid( )` function returns `0` . Otherwise, `–1` is returned and `errno` is set to indicate the error.

ERRORS

The `getauid( )` and `setauid( )` functions will fail if:

`EFAULT` The *auid* argument points to an invalid address.

`EPERM` The process does not have the appropriate privileges.

SUMMARY OF TRUSTED SOLARIS CHANGES

This functionality is active only if auditing is enabled. By default, auditing is enabled in the Trusted Solaris environment. See *Trusted Solaris Audit Administration* for more information.

The privileges explained in `DESCRIPTION` are needed to run this command successfully.

These system calls have been superseded by `getaudit( )` and `setaudit( )` .

SEE ALSO

Trusted Solaris 8 Reference Manual

`audit`(2) , `getaudit`(2)

NAME

setclearance – Set process clearance

SYNOPSIS

cc [*flags*…] *file* … `–ltsol` [*library*…]

```
#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` On success.

`–1` On failure, 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.

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

Trusted Solaris 8 Reference Manual

`getclearance`(2)

NAME

setcmwlabel, fsetcmwlabel, lsetcmwlabel – Set CMW label of a file

SYNOPSIS

cc [*flags*…] *file* … `-ltsol` [*library*…]

```
#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 file system that supports the setting of labels on individual objects.

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 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 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* 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 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 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 On success.

–1 On failure, 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 file system that supports the setting of labels on individual objects. 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(3C)] while _POSIX_NO_TRUNC is in effect. See pathconf(2) .
ENOENT	The file referred to by *path* does not exist.
ENOTDIR	A component of the path prefix of *path* is not a directory.
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. A calling process that is not the owner of the file attempted to downgrade the 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 any of these conditions prevails:

`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 file system that supports the setting of labels on individual objects.
	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.
	The calling process is not the owner of the file, attempted to downgrade the 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.
`EPERM`	The calling process does not have mandatory write access to the object referred to by *fd* because the sensitivity label of the object referred to by *fd* 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.
	A calling process that is not the owner of the file attempted to downgrade the sensitivity label associated with the object referred to by *fd* 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 object referred to by *fd* 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 object referred to by *fd* 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

Trusted Solaris 8 Reference Manual

`getcmwfsrange`(2), `getcmwlabel`(2)

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 sensitivity label or the CMW label for the process making the call. The *flag* argument identifies which label to set:

SETCL_ALL Set the entire CMW label of the process.

SETCL_SL Set only the sensitivity 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 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 value for sensitivity label does not match current value of the process, the set of effective privileges of the calling process must include PRIV_PROC_SETSL.

RETURN VALUES

setcmwplabel( ) returns:

0 On success.

–1 On failure, 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:

EBUSY The process is being accessed through the /proc filesystem, which can happen when the process is either being traced or debugged, and the caller of setcmwplabel( ) lacks the PRIV_PROC_TRANQUIL privilege.

The *label_p* argument points to an invalid address.

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 sensitivity label not to be dominated by the clearance.

EPERM The calling process lacks the PRIV_PROC_SETSL privilege necessary to set the sensitivity label specified by *flag*.

SEE ALSO

Trusted Solaris 8 Reference Manual

`getcmwplabel`(2)

SunOS 5.8 Reference Manual

`attributes`(5)

NAME

setuid, setegid, seteuid, setgid – Set user and group IDs

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
int setuid(uid_t uid);

int setegid(gid_t egid);

int seteuid(uid_t euid);

int setgid(gid_t gid);
```

DESCRIPTION

The `setuid()` function sets the real user ID , effective user ID , and saved user ID of the calling process. The `setgid()` function sets the real group ID , effective group ID , and saved group ID of the calling process. The `setegid()` and `seteuid()` functions set the effective group and user ID s respectively for the calling process. See `intro`(2) for more information on real, effective, and saved user and group ID s.

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 ID s; they are set to the group ID of the user responsible for the creation of the process.

When a process calls one of the `exec` family of functions (see `exec`(2)) to execute a file (program), the user and / or group identifiers associated with the process can change. If the file executed is a set-user- ID file, the effective and saved user ID s 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 ID s 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 process calling `setgid()` has the `PRIV_PROC_SETID` privilege, the real, effective, and saved group ID s 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

`setuid()` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `setuid()` and `setgid()` functions will fail if:

`EINVAL` The value of *uid* or *gid* is out of range.

`EPERM` For `setuid()` and `seteuid()` , the 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 ID s.

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 .

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	`setuid()` and `setgid()` are Async-Signal-Safe

SUMMARY OF TRUSTED SOLARIS CHANGES

The Trusted Solaris environment replaces the checks of super-user in the Solaris environment with a check for `PRIV_PROC_SETID` .

SEE ALSO

Trusted Solaris 8 Reference Manual

`intro`(2) , `exec`(2) , `getgroups`(2)

SunOS 5.8 Reference Manual

`getuid`(2) , `attributes`(5) , `stat`(5)

NAME

setuid, setegid, seteuid, setgid – Set user and group IDs

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
int setuid(uid_t uid);

int setegid(gid_t egid);

int seteuid(uid_t euid);

int setgid(gid_t gid);
```

DESCRIPTION

The `setuid()` function sets the real user ID , effective user ID , and saved user ID of the calling process. The `setgid()` function sets the real group ID , effective group ID , and saved group ID of the calling process. The `setegid()` and `seteuid()` functions set the effective group and user ID s respectively for the calling process. See `intro`(2) for more information on real, effective, and saved user and group ID s.

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 ID s; they are set to the group ID of the user responsible for the creation of the process.

When a process calls one of the `exec` family of functions (see `exec`(2)) to execute a file (program), the user and / or group identifiers associated with the process can change. If the file executed is a set-user- ID file, the effective and saved user ID s 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 ID s 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 process calling `setgid()` has the `PRIV_PROC_SETID` privilege, the real, effective, and saved group ID s 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

`setuid()` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `setuid()` and `setgid()` functions will fail if:

`EINVAL` The value of *uid* or *gid* is out of range.

`EPERM` For `setuid()` and `seteuid()` , the 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 ID s.

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 .

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	`setuid()` and `setgid()` are Async-Signal-Safe

SUMMARY OF TRUSTED SOLARIS CHANGES

The Trusted Solaris environment replaces the checks of super-user in the Solaris environment with a check for `PRIV_PROC_SETID` .

SEE ALSO

Trusted Solaris 8 Reference Manual

`intro`(2) , `exec`(2) , `getgroups`(2)

SunOS 5.8 Reference Manual

`getuid`(2) , `attributes`(5) , `stat`(5)

NAME

getfattrflag, fsetfattrflag, fgetfattrflag, setfattrflag, mldgetfattrflag, mldsetfattrflag – Set / get the security attribute flags of a file

SYNOPSIS

cc [*flags*…] *file*… –ltsol

#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* .

RETURN VALUES

These functions return:

`0` On success.

`–1` On failure, and set `errno` 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* .
`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 filesystem 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 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

Trusted Solaris 8 Reference Manual

`setfattrflag`(1) , `getfattrflag`(1)

Trusted Solaris Developer's Guide

NAME

getfpriv, fgetfpriv, setfpriv, fsetfpriv – Return or set a privilege set associated with a file

SYNOPSIS

cc [*flags*…] *file*… –ltsol

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 `PRIV_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 `PRIV_FILE_SETPRIV` must be effective. In addition, only the owner of a file may change its privilege sets, unless the privilege `PRIV_FILE_OWNER` is effective.

The invoking process must have MAC write permission for the named file (unless the privilege `PRIV_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.

Normally MAC read permission is required or the privilege PRIV_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 PRIV_FILE_MAC_READ is not effective then getfpriv( ) returns success but sets the privilege fields of *priv_set* to zero.

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 PRIV_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 PRIV_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 (that is, a file system other than NFS , TMPFS) or (2) an illegal or undefined value is supplied for *op* . Also if privilege PRIV_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) PRIV_FILE_SETPRIV is not effective, or (3) the effective uid does not match the owner of the named file and privilege PRIV_FILE_OWNER is not effective.
EROFS	The named file resides on a read-only file system.

SEE ALSO

Trusted Solaris 8 Reference Manual

getppriv(2) , setppriv(2) , priv_macros(5)

SunOS 5.8 Reference Manual

attributes(5)

NAME

setuid, setegid, seteuid, setgid – Set user and group IDs

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
int setuid(uid_t uid);

int setegid(gid_t egid);

int seteuid(uid_t euid);

int setgid(gid_t gid);
```

DESCRIPTION

The `setuid()` function sets the real user ID , effective user ID , and saved user ID of the calling process. The `setgid()` function sets the real group ID , effective group ID , and saved group ID of the calling process. The `setegid()` and `seteuid()` functions set the effective group and user ID s respectively for the calling process. See `intro`(2) for more information on real, effective, and saved user and group ID s.

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 ID s; they are set to the group ID of the user responsible for the creation of the process.

When a process calls one of the `exec` family of functions (see `exec`(2)) to execute a file (program), the user and / or group identifiers associated with the process can change. If the file executed is a set-user- ID file, the effective and saved user ID s 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 ID s 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 process calling `setgid()` has the `PRIV_PROC_SETID` privilege, the real, effective, and saved group ID s 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

`setuid()` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `setuid()` and `setgid()` functions will fail if:

`EINVAL` The value of *uid* or *gid* is out of range.

`EPERM` For `setuid()` and `seteuid()` , the 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 ID s.

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 .

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	`setuid()` and `setgid()` are Async-Signal-Safe

SUMMARY OF TRUSTED SOLARIS CHANGES

The Trusted Solaris environment replaces the checks of super-user in the Solaris environment with a check for `PRIV_PROC_SETID` .

SEE ALSO

Trusted Solaris 8 Reference Manual

`intro`(2) , `exec`(2) , `getgroups`(2)

SunOS 5.8 Reference Manual

`getuid`(2) , `attributes`(5) , `stat`(5)

NAME

getgroups, setgroups – Get or set supplementary group access list IDs

SYNOPSIS

```
#include <unistd.h>
int getgroups(int gidsetsize, gid_t * grouplist);

int setgroups(int ngroups, const gid_t * grouplist);
```

DESCRIPTION

The `getgroups()` function gets the current supplemental group access list of the calling process and stores the result in the array of group ID s specified by *grouplist* . This array has *gidsetsize* entries and must be large enough to contain the entire list. This list cannot be larger than `NGROUPS_MAX` . If *gidsetsize* equals 0, `getgroups()` will return the number of groups to which the calling process belongs without modifying the array pointed to by *grouplist* .

The `setgroups()` function sets the supplementary group access list of the calling process from the array of group ID s specified by *grouplist* . The number of entries is specified by *ngroups* and can not 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 ID s set for the calling process and `setgroups()` returns `0` . Otherwise, `–1` is returned and `errno` is set to indicate the error.

ERRORS

The `getgroups()` and `setgroups()` functions will fail if:

`EFAULT` A referenced part of the array pointed to by *grouplist* is an illegal address.

The `getgroups()` function will fail if:

`EINVAL` The value of *gidsetsize* is non-zero and less than the number of supplementary group ID s set for the calling process.

The `setgroups()` function will fail if:

`EINVAL` The value of *ngroups* is greater than `NGROUPS_MAX` .

`EPERM` The calling process does not have the `PRIV_PROC_SETID` privilege.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

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

Trusted Solaris 8 Reference Manual

`chown`(2) , `setuid`(2)

SunOS 5.8 Reference Manual

`groups`(1) , `getuid`(2) , `getgrnam`(3C) , `initgroups`(3C) , `attributes`(5)

NAME

getpattr, setpattr – Get / set process attribute flags

SYNOPSIS

cc [*flags*…] *file*… –ltsol

#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 , PAF_TP_OFF) 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.

```
setpattr(PAF_TRUSTED_PATH, PAF_TP_OFF)
```

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
`getpattr()` and `setpattr()` return:

`0` On success.

`–1` On failure, 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
Trusted Solaris 8 Reference Manual

`pattr`(1)

NAME

getppriv, setppriv – Return or assign a privilege set associated with the invoking process

SYNOPSIS

cc [*flags*…] *file*… –ltsol

```
#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`	On success.
`–1`	On failure, 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

Trusted Solaris 8 Reference Manual

`getfpriv`(2) , `setfpriv`(2) , `priv_to_str`(3TSOL) , `priv_set_to_str`(3TSOL) , `str_to_priv`(3TSOL) , `str_to_priv_set`(3TSOL) , `priv_macros`(5)

NAME

setregid – Set real and effective group IDs

SYNOPSIS

```
#include <unistd.h>
int setregid(gid_t rgid, gid_t egid);
```

DESCRIPTION

The `setregid()` function is used to set the real and effective group IDs of the calling process. If *rgid* is –1, the real group ID is not changed; if *egid* is –1, the effective group ID is not changed. The real and effective group IDs may be set to different values in the same call.

If the 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 group ID is being changed (that is, if *rgid* is not –1), or the effective group ID is being changed to a value not equal to the real group ID, the saved set-group-ID is set equal to the new effective group ID.

RETURN VALUES

`setregid()` returns:

`0` On success.

`–1` On failure and sets `errno` to indicate the error.

ERRORS

The `setregid()` function will fail if:

`EINVAL` The value of *rgid* or *egid* is less than 0 or greater than `UID_MAX` (defined in `<limits.h>`).

`EPERM` The 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.

USAGE

If a set-group-ID process sets its effective group ID to its real group ID, it can still set its effective group ID back to the saved set-group- ID.

SUMMARY OF TRUSTED SOLARIS CHANGES

The Trusted Solaris environment replaces the checks of super-user in the Solaris environment with privilege checks.

SEE ALSO

Trusted Solaris 8 Reference Manual

`execve`(2), `setreuid`(2), `setuid`(2)

SunOS 5.8 Reference Manual

`getgid`(2)

NAME

setreuid – Set real and effective user IDs

SYNOPSIS

```
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
```

DESCRIPTION

The `setreuid( )` function is used to set the real and effective user IDs of the calling process. If *ruid* is –1, the real user ID is not changed; if *euid* is –1, the effective user ID is not changed. The real and effective user IDs may be set to different values in the same call.

If the 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

The `setreuid( )` function will fail if:

`EINVAL` The value of *ruid* or *euid* is less than `0` or greater than `UID_MAX` (defined in `<limits.h>`).

`EPERM` The calling process 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.

USAGE

If a set-user-ID process sets its effective user ID to its real user ID, it can still set its effective user ID back to the saved set-user ID.

SUMMARY OF TRUSTED SOLARIS CHANGES

The Trusted Solaris environment replaces the checks of super-user in the Solaris environment with privilege checks.

SEE ALSO
Trusted Solaris 8 Reference Manual

`execve`(2), `setregid`(2), `setuid`(2)

SunOS 5.8 Reference Manual

`getuid`(2)

NAME

getrlimit, setrlimit – Control maximum system resource consumption

SYNOPSIS

```
#include <sys/resource.h>
int getrlimit(int resource, struct rlimit * rlp);

int setrlimit(int resource, const struct rlimit * rlp);
```

DESCRIPTION

Limits on the consumption of a variety of system resources by a process and each process it creates may be obtained with the `getrlimit( )` and set with `setrlimit( )` functions.

Each call to either `getrlimit( )` or `setrlimit( )` identifies a specific resource to be operated upon as well as a resource limit. A resource limit is a pair of values: one specifying the current (soft) limit, the other a maximum (hard) limit. Soft limits may be changed by a process to any value that is less than or equal to the hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or equal to the soft limit. Only a process 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` . The *rlp* argument is a pointer to `struct rlimit` that includes the following members:

```
rlim_t    rlim_cur;     /* current (soft) limit */
rlim_t    rlim_max;     /* hard limit */
```

The type `rlim_t` is an arithmetic data type to which objects of type `int` , `size_t` , and `off_t` can be cast without loss of information.

The possible resources, their descriptions, and the actions taken when the current limit is exceeded are summarized as follows:

`RLIMIT_CORE` The maximum size of a core file in bytes that may be created by a process. A limit of `0` will prevent the creation of a core file. The writing of a core file will terminate at this size.

`RLIMIT_CPU` The maximum amount of CPU time in seconds used by a process. This is a soft limit only. The `SIGXCPU` signal is sent to the process. If the process is holding or ignoring `SIGXCPU` , the behavior is scheduling class defined.

`RLIMIT_DATA` The maximum size of a process's heap in bytes. The `brk`(2) function will fail with `errno` set to `ENOMEM` .

`RLIMIT_FSIZE` The maximum size of a file in bytes that may be created by a process. A limit of `0` will prevent the creation of a file. The `SIGXFSZ` signal is sent to the process. If the process is holding or ignoring `SIGXFSZ` , continued attempts to

increase the size of a file beyond the limit will fail with `errno` set to `EFBIG` .

`RLIMIT_NOFILE` One more than the maximum value that the system may assign to a newly created descriptor. This limit constrains the number of file descriptors that a process may create.

`RLIMIT_STACK` The maximum size of a process's stack in bytes. The system will not automatically grow the stack beyond this limit.

Within a process, `setrlimit( )` will increase the limit on the size of your stack, but will not move current memory segments to allow for that growth. To guarantee that the process stack can grow to the limit, the limit must be altered prior to the execution of the process in which the new stack size is to be used.

Within a multithreaded process, `setrlimit( )` has no impact on the stack size limit for the calling thread if the calling thread is not the main thread. A call to `setrlimit( )` for `RLIMIT_STACK` impacts only the main thread's stack, and should be made only from the main thread, if at all.

The `SIGSEGV` signal is sent to the process. If the process is holding or ignoring `SIGSEGV` , or is catching `SIGSEGV` and has not made arrangements to use an alternate stack (see `sigaltstack`(2)), the disposition of `SIGSEGV` will be set to `SIG_DFL` before it is sent.

`RLIMIT_VMEM` The maximum size of a process's mapped address space in bytes. If this limit is exceeded, the `brk`(2) and `mmap`(2) functions will fail with `errno` set to `ENOMEM` . In addition, the automatic stack growth will fail with the effects outlined above.

`RLIMIT_AS` This is the maximum size of a process's total available memory, in bytes. If this limit is exceeded, the `brk`(2) , `malloc`(3C) , `mmap`(2) and `sbrk`(2) functions will fail with `errno` set to `ENOMEM` . In addition, the automatic stack growth will fail with the effects outlined above.

Because limit information is stored in the per-process information, the shell builtin `ulimit` command must directly execute this system call if it is to affect all future processes created by the shell.

The value of the current limit of the following resources affect these implementation defined parameters:

Limit	Implementation Defined Constant
RLIMIT_FSIZE	FCHR_MAX
RLIMIT_NOFILE	OPEN_MAX

When using the `getrlimit()` function, if a resource limit can be represented correctly in an object of type `rlim_t`, then its representation is returned; otherwise, if the value of the resource limit is equal to that of the corresponding saved hard limit, the value returned is `RLIM_SAVED_MAX`; otherwise the value returned is `RLIM_SAVED_CUR`.

When using the `setrlimit()` function, if the requested new limit is `RLIM_INFINITY`, the new limit will be "no limit"; otherwise if the requested new limit is `RLIM_SAVED_MAX`, the new limit will be the corresponding saved hard limit; otherwise, if the requested new limit is `RLIM_SAVED_CUR`, the new limit will be the corresponding saved soft limit; otherwise, the new limit will be the requested value. In addition, if the corresponding saved limit can be represented correctly in an object of type `rlim_t`, then it will be overwritten with the new limit.

The result of setting a limit to `RLIM_SAVED_MAX` or `RLIM_SAVED_CUR` is unspecified unless a previous call to `getrlimit()` returned that value as the soft or hard limit for the corresponding resource limit.

A limit whose value is greater than `RLIM_INFINITY` is permitted.

The `exec` family of functions also cause resource limits to be saved. See `exec`(2).

RETURN VALUES

`getrlimit()` and `setrlimit()` return:

`0` On success.

`–1` On failure, and set `errno` to indicate the error.

ERRORS

The `getrlimit()` and `setrlimit()` functions will fail if:

`EFAULT` The *rlp* argument points to an illegal address.

`EINVAL` An invalid *resource* was specified; or in a `setrlimit()` call, the new `rlim_cur` exceeds the new `rlim_max`.

`EPERM` The limit specified to `setrlimit()` would have raised the maximum limit value, and the calling process does not have the `PRIV_SYS_CONFIG` privilege.

The `setrlimit()` function may fail if:

`EINVAL` The limit specified cannot be lowered because current usage is already higher than the limit.

USAGE

The `getrlimit( )` and `setrlimit( )` functions have transitional interfaces for 64-bit file offsets. See `lf64`(5) .

SUMMARY OF TRUSTED SOLARIS CHANGES

The calling process must have the `PRIV_SYS_CONFIG` privilege in order to increase a hard resource limit.

SEE ALSO

Trusted Solaris 8 Reference Manual

`open`(2)

SunOS 5.8 Reference Manual

`brk`(2) , `sigaltstack`(2) , `malloc`(3C) , `signal`(3C) , `signal`(5)

NAME

setuid, setegid, seteuid, setgid – Set user and group IDs

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
int setuid(uid_t uid);

int setegid(gid_t egid);

int seteuid(uid_t euid);

int setgid(gid_t gid);
```

DESCRIPTION

The `setuid( )` function sets the real user ID , effective user ID , and saved user ID of the calling process. The `setgid( )` function sets the real group ID , effective group ID , and saved group ID of the calling process. The `setegid( )` and `seteuid( )` functions set the effective group and user ID s respectively for the calling process. See `intro`(2) for more information on real, effective, and saved user and group ID s.

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 ID s; they are set to the group ID of the user responsible for the creation of the process.

When a process calls one of the `exec` family of functions (see `exec`(2)) to execute a file (program), the user and / or group identifiers associated with the process can change. If the file executed is a set-user- ID file, the effective and saved user ID s 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 ID s 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 process calling `setgid( )` has the `PRIV_PROC_SETID` privilege, the real, effective, and saved group ID s 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

`setuid()` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `setuid()` and `setgid()` functions will fail if:

`EINVAL` The value of *uid* or *gid* is out of range.

`EPERM` For `setuid()` and `seteuid()` , the 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 ID s.

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 .

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	`setuid()` and `setgid()` are Async-Signal-Safe

SUMMARY OF TRUSTED SOLARIS CHANGES

The Trusted Solaris environment replaces the checks of super-user in the Solaris environment with a check for `PRIV_PROC_SETID` .

SEE ALSO

Trusted Solaris 8 Reference Manual

`intro`(2) , `exec`(2) , `getgroups`(2)

SunOS 5.8 Reference Manual

`getuid`(2) , `attributes`(5) , `stat`(5)

NAME

shmop, shmat, shmdt – Shared memory operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/shm.h>
void * shmat(int shmid, const void * shmaddr, int shmflg);
```

Default

```
int shmdt(char * shmaddr);
```

Standard-conforming

```
int shmdt(const void * shmaddr);
```

DESCRIPTION

The `shmat()` function attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process.

The permission required for a shared memory control operation is given as { *token* }, where *token* is the type of permission needed. The types of permission are interpreted as follows:

```
00400    READ by user
00200    WRITE by user
00040    READ by group
00020    WRITE by group
00004    READ by others
00002    WRITE by others
```

See the *Shared Memory Operation Permissions* section of `intro`(2) for more information.

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 the following criteria:

- If *shmaddr* is equal to `(void *) 0` , the segment is attached to the first available address as selected by the system.

- If *shmaddr* is equal to `(void *) 0` and (*shmflg* &SHM_SHARE_MMU) is true, then the segment is attached to the first available suitably aligned address. When (*shmflg* &SHM_SHARE_MMU) is set, however, the permission given by `shmget( )` determines whether the segment is attached for reading or reading and writing.
- If *shmaddr* is not equal to `(void *) 0` and (*shmflg* &SHM_RND) is true, the segment is attached to the address given by (*shmaddr* - (*shmaddr* modulus SHMLBA)).
- If *shmaddr* is not equal to `(void *) 0` and (*shmflg* &SHM_RND) is false, the segment is attached to the address given by *shmaddr* .

The segment is attached for reading if (*shmflg* &SHM_RDONLY) is true { READ }, otherwise it is attached for reading and writing { READ/WRITE }.

The `shmdt( )` function detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr* . If the application is standard-conforming [see `standards`(5)], the *shmaddr* argument is of type `const void *` . Otherwise it is of type `char *` .

Shared memory segments must be explicitly removed after the last reference to them has been removed.

RETURN VALUES

Upon successful completion, `shmat( )` returns the data segment start address of the attached shared memory segment; `shmdt( )` returns `0` . Otherwise, `–1` is returned, the shared memory segment is not attached, and `errno` is set to indicate the error.

ERRORS

The `shmat( )` function will fail if:

EACCES	Operation permission is denied to the calling process [see `intro`(2)], and the calling process does not have the appropriate privilege(s) in its set of effective privileges.
EINVAL	The *shmid* argument is not a valid shared memory identifier.
EINVAL	The *shmaddr* argument is not equal to 0, and the value of (*shmaddr* - (*shmaddr* modulus SHMLBA)) is an illegal address.
EINVAL	The *shmaddr* argument is not equal to 0, is an illegal address, and (*shmflg* &SHM_RND) is false.
EINVAL	The *shmaddr* argument is not equal to 0, is not properly aligned, and (*shmfg* &SHM_SHARE_MMU) is true.
EINVAL	SHM_SHARE_MMU is not supported in certain architectures.
EMFILE	The number of shared memory segments attached to the calling process would exceed the system-imposed limit.

ENOMEM	The available data space is not large enough to accommodate the shared memory segment.
EPERM	The LOCK and UNLOCK operation does not have the appropriate privilege in its set of effective privileges.

The shmdt( ) function will fail if:

EINVAL	The *shmaddr* argument is not the data segment start address of a shared memory segment.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

Trusted Solaris 8 Reference Manual

intro(2) , exec(2) , fork(2) , shmctl(2) , shmget(2)

SunOS 5.8 Reference Manual

exit(2) , intro(2) , xpg4(5)

NAME

shmctl – Shared memory control operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

DESCRIPTION

The `shmctl( )` function provides a variety of shared memory control operations as specified by *cmd*. The permission required for a shared memory control operation is given as {*token*}, where *token* is the type of permission needed. The types of permission are interpreted as follows:

```
00400    READ by user
00200    WRITE by user
00040    READ by group
00020    WRITE by group
00004    READ by others
00002    WRITE by others
```

See the *Shared Memory Operation Permissions* section of `intro`(2) for more information.

The following operations require the specified tokens:

`IPC_STAT` Place the current value of each member of the data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure are defined in `intro`(2) {READ}

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 pointed to by *buf*:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode /* access permission bits only */
```

This command can be executed only by a process that 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.

Shared memory segments must be explicitly removed after the last reference to them has been removed.

RETURN VALUES

shmctl( ) returns:

0 On success.

–1 On failure, and sets errno to indicate the error.

ERRORS

The shmctl( ) function will fail if:

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 — The *buf* argument points to an illegal address.

EINVAL — The *shmid* argument is not a valid shared memory identifier; or the *cmd* argument is not a valid command or is IPC_SET and shm_perm.uid or shm_perm.gid is not valid.

ENOMEM — The *cmd* argument is equal to SHM_LOCK and there is not enough memory.

EOVERFLOW	The *cmd* argument is IPC_STAT and *uid* or *gid* is too large to be stored in the structure pointed to by *buf*.
EPERM	*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

Trusted Solaris 8 Reference Manual

ipcs(1), intro(2), shmget(2), shmop(2)

NAME

shmop, shmat, shmdt – Shared memory operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/shm.h>
void * shmat(int shmid, const void * shmaddr, int shmflg);
```

Default

```
int shmdt(char * shmaddr);
```

Standard-conforming

```
int shmdt(const void * shmaddr);
```

DESCRIPTION

The `shmat()` function attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process.

The permission required for a shared memory control operation is given as { *token* }, where *token* is the type of permission needed. The types of permission are interpreted as follows:

```
00400    READ by user
00200    WRITE by user
00040    READ by group
00020    WRITE by group
00004    READ by others
00002    WRITE by others
```

See the *Shared Memory Operation Permissions* section of `intro`(2) for more information.

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 the following criteria:

- If *shmaddr* is equal to `(void *) 0` , the segment is attached to the first available address as selected by the system.

- If *shmaddr* is equal to `(void *) 0` and (*shmflg* &SHM_SHARE_MMU) is true, then the segment is attached to the first available suitably aligned address. When (*shmflg* &SHM_SHARE_MMU) is set, however, the permission given by `shmget()` determines whether the segment is attached for reading or reading and writing.
- If *shmaddr* is not equal to `(void *) 0` and (*shmflg* &SHM_RND) is true, the segment is attached to the address given by (*shmaddr* - (*shmaddr* modulus SHMLBA)).
- If *shmaddr* is not equal to `(void *) 0` and (*shmflg* &SHM_RND) is false, the segment is attached to the address given by *shmaddr* .

The segment is attached for reading if (*shmflg* &SHM_RDONLY) is true { READ }, otherwise it is attached for reading and writing { READ/WRITE }.

The `shmdt()` function detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr* . If the application is standard-conforming [see `standards`(5)], the *shmaddr* argument is of type `const void *` . Otherwise it is of type `char *` .

Shared memory segments must be explicitly removed after the last reference to them has been removed.

RETURN VALUES

Upon successful completion, `shmat()` returns the data segment start address of the attached shared memory segment; `shmdt()` returns `0` . Otherwise, `–1` is returned, the shared memory segment is not attached, and `errno` is set to indicate the error.

ERRORS

The `shmat()` function will fail if:

EACCES	Operation permission is denied to the calling process [see `intro`(2)], and the calling process does not have the appropriate privilege(s) in its set of effective privileges.
EINVAL	The *shmid* argument is not a valid shared memory identifier.
EINVAL	The *shmaddr* argument is not equal to 0, and the value of (*shmaddr* - (*shmaddr* modulus SHMLBA)) is an illegal address.
EINVAL	The *shmaddr* argument is not equal to 0, is an illegal address, and (*shmflg* &SHM_RND) is false.
EINVAL	The *shmaddr* argument is not equal to 0, is not properly aligned, and (*shmfg* &SHM_SHARE_MMU) is true.
EINVAL	SHM_SHARE_MMU is not supported in certain architectures.
EMFILE	The number of shared memory segments attached to the calling process would exceed the system-imposed limit.

ENOMEM	The available data space is not large enough to accommodate the shared memory segment.
EPERM	The LOCK and UNLOCK operation does not have the appropriate privilege in its set of effective privileges.

The shmdt( ) function will fail if:

EINVAL	The *shmaddr* argument is not the data segment start address of a shared memory segment.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

Trusted Solaris 8 Reference Manual

intro(2) , exec(2) , fork(2) , shmctl(2) , shmget(2)

SunOS 5.8 Reference Manual

exit(2) , intro(2) , xpg4(5)

NAME

shmget, shmgetl – Get shared memory segment identifier

SYNOPSIS

cc [*flags*…] *file* … –ltsol [*library*…]

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);

#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 specified 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` .

The `shmget()` function returns the shared memory identifier associated with *key* .

A shared memory identifier and associated data structure and shared memory segment of at least *size* bytes (see `intro`(3)) are created for *key* if one of the following are true:

- The *key* argument is equal to `IPC_PRIVATE` .
- The *key* argument does not already have a shared memory identifier associated with it, and (*shmflg* `&IPC_CREAT`) is true.

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

- The values of `shm_perm.cuid` , `shm_perm.uid` , `shm_perm.cgid` , and `shm_perm.gid` are set equal to the effective user ID and effective group ID , respectively, of the calling process.
- The access permission bits of `shm_perm.mode` are set equal to the access permission bits of *shmflg* . `shm_segsz` is set equal to the value of *size* .

- The values of `shm_lpid` , `shm_nattch` `shm_atime` , and `shm_dtime` are set equal to 0.
- The `shm_ctime` is set equal to the current time.

Shared memory segments must be explicitly removed after the last reference to them has been removed.

RETURN VALUES

Upon successful completion, a non-negative integer representing a shared memory identifier is returned. Otherwise, `–1` is returned and `errno` is set to indicate the error.

ERRORS

The `shmget( )` function will fail if:

`EACCES` A shared memory identifier exists for the union of key and sensitivity label, but operation permission (see `intro`(3)) as specified by the low-order 9 bits of *shmflg* 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` The *size* argument is less than the system-imposed minimum or greater than the system-imposed maximum.

`EINVAL` A shared memory identifier exists for *key* but the size of the segment associated with it is less than *size* and *size* is not equal to 0.

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 system-wide 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

Trusted Solaris 8 Reference Manual

`shmctl`(2) , `shmop`(2) , `intro`(3)

SunOS 5.8 Reference Manual

`ftok`(3C)

NAME

shmget, shmgetl – Get shared memory segment identifier

SYNOPSIS

cc [*flags*…] *file* … –`ltsol` [*library*…]

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);

#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 specified 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` .

The `shmget( )` function returns the shared memory identifier associated with *key* .

A shared memory identifier and associated data structure and shared memory segment of at least *size* bytes (see `intro`(3)) are created for *key* if one of the following are true:

- The *key* argument is equal to `IPC_PRIVATE` .
- The *key* argument does not already have a shared memory identifier associated with it, and (*shmflg* &`IPC_CREAT`) is true.

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

- The values of `shm_perm.cuid` , `shm_perm.uid` , `shm_perm.cgid` , and `shm_perm.gid` are set equal to the effective user ID and effective group ID , respectively, of the calling process.
- The access permission bits of `shm_perm.mode` are set equal to the access permission bits of *shmflg* . `shm_segsz` is set equal to the value of *size* .

- The values of `shm_lpid` , `shm_nattch` `shm_atime` , and `shm_dtime` are set equal to 0.
- The `shm_ctime` is set equal to the current time.

Shared memory segments must be explicitly removed after the last reference to them has been removed.

RETURN VALUES

Upon successful completion, a non-negative integer representing a shared memory identifier is returned. Otherwise, `–1` is returned and `errno` is set to indicate the error.

ERRORS

The `shmget( )` function will fail if:

`EACCES` A shared memory identifier exists for the union of key and sensitivity label, but operation permission (see `intro`(3)) as specified by the low-order 9 bits of *shmflg* 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` The *size* argument is less than the system-imposed minimum or greater than the system-imposed maximum.

`EINVAL` A shared memory identifier exists for *key* but the size of the segment associated with it is less than *size* and *size* is not equal to 0.

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 system-wide 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

Trusted Solaris 8 Reference Manual

`shmctl`(2) , `shmop`(2) , `intro`(3)

SunOS 5.8 Reference Manual

`ftok`(3C)

NAME

shmop, shmat, shmdt – Shared memory operations

SYNOPSIS

```
#include <sys/types.h>
#include <sys/shm.h>
void * shmat(int shmid, const void * shmaddr, int shmflg);
```

Default

int **shmdt**(char * *shmaddr*);

Standard-conforming

int **shmdt**(const void * *shmaddr*);

DESCRIPTION

The `shmat()` function attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the data segment of the calling process.

The permission required for a shared memory control operation is given as { *token* }, where *token* is the type of permission needed. The types of permission are interpreted as follows:

```
00400    READ by user
00200    WRITE by user
00040    READ by group
00020    WRITE by group
00004    READ by others
00002    WRITE by others
```

See the *Shared Memory Operation Permissions* section of `intro`(2) for more information.

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 the following criteria:

- If *shmaddr* is equal to `(void *) 0` , the segment is attached to the first available address as selected by the system.

- If *shmaddr* is equal to `(void *) 0` and (*shmflg* &SHM_SHARE_MMU) is true, then the segment is attached to the first available suitably aligned address. When (*shmflg* &SHM_SHARE_MMU) is set, however, the permission given by `shmget( )` determines whether the segment is attached for reading or reading and writing.
- If *shmaddr* is not equal to `(void *) 0` and (*shmflg* &SHM_RND) is true, the segment is attached to the address given by (*shmaddr* - (*shmaddr* modulus SHMLBA)).
- If *shmaddr* is not equal to `(void *) 0` and (*shmflg* &SHM_RND) is false, the segment is attached to the address given by *shmaddr* .

The segment is attached for reading if (*shmflg* &SHM_RDONLY) is true { READ }, otherwise it is attached for reading and writing { READ/WRITE }.

The `shmdt( )` function detaches from the calling process's data segment the shared memory segment located at the address specified by *shmaddr* . If the application is standard-conforming [see `standards`(5)], the *shmaddr* argument is of type `const void *` . Otherwise it is of type `char *` .

Shared memory segments must be explicitly removed after the last reference to them has been removed.

RETURN VALUES

Upon successful completion, `shmat( )` returns the data segment start address of the attached shared memory segment; `shmdt( )` returns `0` . Otherwise, `–1` is returned, the shared memory segment is not attached, and `errno` is set to indicate the error.

ERRORS

The `shmat( )` function will fail if:

EACCES	Operation permission is denied to the calling process [see `intro`(2)], and the calling process does not have the appropriate privilege(s) in its set of effective privileges.
EINVAL	The *shmid* argument is not a valid shared memory identifier.
EINVAL	The *shmaddr* argument is not equal to 0, and the value of (*shmaddr* - (*shmaddr* modulus SHMLBA)) is an illegal address.
EINVAL	The *shmaddr* argument is not equal to 0, is an illegal address, and (*shmflg* &SHM_RND) is false.
EINVAL	The *shmaddr* argument is not equal to 0, is not properly aligned, and (*shmfg* &SHM_SHARE_MMU) is true.
EINVAL	SHM_SHARE_MMU is not supported in certain architectures.
EMFILE	The number of shared memory segments attached to the calling process would exceed the system-imposed limit.

ENOMEM The available data space is not large enough to accommodate the shared memory segment.

EPERM The LOCK and UNLOCK operation does not have the appropriate privilege in its set of effective privileges.

The shmdt() function will fail if:

EINVAL The *shmaddr* argument is not the data segment start address of a shared memory segment.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

Trusted Solaris 8 Reference Manual

intro(2) , exec(2) , fork(2) , shmctl(2) , shmget(2)

SunOS 5.8 Reference Manual

exit(2) , intro(2) , xpg4(5)

NAME

sigsend, sigsendset – Send a signal to a process or a group of processes

SYNOPSIS

```
#include <signal.h>
int sigsend(idtype_t idtype, id_t id, int sig);

int sigsendset(procset_t * psp, int sig);
```

DESCRIPTION

The `sigsend( )` function sends a signal to the process or group of processes specified by *id* and *idtype* . The signal to be sent is specified by *sig* and is either 0 or one of the values listed in `signal`(3HEAD) . If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This value can be used to check the validity of *id* and *idtype* .

The 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 all process with process group ID *id* .

If *idtype* is `P_SID` , *sig* is sent to all process with session ID *id* .

If *idtype* is `P_UID` , *sig* is sent to any process with effective user ID *id* .

If *idtype* is `P_GID` , *sig* is sent to any process with effective group ID *id* .

If *idtype* is `P_CID` , *sig* is sent to any process with scheduler class ID *id* (see `priocntl`(2)).

If *idtype* is `P_ALL` , *sig* is sent to all processes and *id* is ignored.

If *id* is `P_MYID` , the value of *id* is taken from the calling process.

The process with a process ID of 0 is always excluded. The process with a process ID of 1 is excluded unless *idtype* is equal to `P_PID` .

The `sigsendset( )` function provides an alternate interface for sending signals to sets of processes. This function sends signals to the set of processes specified by *psp* . *psp* is a pointer to a structure of type `procset_t` , defined in `<sys/procset.h>` , which includes the following members:

```
idop_t    p_op;
idtype_t  p_lidtype;
id_t      p_lid;
idtype_t  p_ridtype;
id_t      p_rid;
```

The `p_lidtype` and `p_lid` members specify the ID type and ID of one (“left”) set of processes; the `p_ridtype` and `p_rid` members specify the ID type

and ID of a second (“right”) set of processes. ID types and ID s are specified just as for the *idtype* and *id* arguments to `sigsend( )` . The `p_op` member specifies the operation to be performed on the two sets of processes to get the set of processes the function is to apply to. The valid values for `p_op` and the processes they specify are:

`POP_DIFF`	Set difference: processes in left set and not in right set.
`POP_AND`	Set intersection: processes in both left and right sets.
`POP_OR`	Set union: processes in either left or right set or both.
`POP_XOR`	Set exclusive-or: processes in left or right set but not in both.

RETURN VALUES

Upon successful completion, `0` is returned. Otherwise, `–1` is returned and `errno` is set to indicate the error.

ERRORS

The `sigsend( )` and `sigsendset( )` functions will fail if:

`EINVAL`	The *sig* argument is not a valid signal number, or the *idtype* argument is not a valid idtype field.
`EINVAL`	The *sig* argument is `SIGKILL,` *idtype* is `P_PID` and *id* is `1` (proc1).
`EPERM`	The 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 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.

The `sigsendset( )` function will fail if:

`EFAULT`	The *psp* argument 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

Trusted Solaris 8 Reference Manual

`getpid`(2) , `kill`(2) , `priocntl`(2)

SunOS 5.8 Reference Manual

`kill`(1) , `signal`(3C) , `signal`(3HEAD)

NAME

sigsend, sigsendset – Send a signal to a process or a group of processes

SYNOPSIS

#include <signal.h>
int **sigsend**(idtype_t *idtype*, id_t *id*, int *sig*);

int **sigsendset**(procset_t * *psp*, int *sig*);

DESCRIPTION

The sigsend( ) function sends a signal to the process or group of processes specified by *id* and *idtype* . The signal to be sent is specified by *sig* and is either 0 or one of the values listed in signal(3HEAD) . If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This value can be used to check the validity of *id* and *idtype* .

The 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 all process with process group ID *id* .

If *idtype* is P_SID , *sig* is sent to all process with session ID *id* .

If *idtype* is P_UID , *sig* is sent to any process with effective user ID *id* .

If *idtype* is P_GID , *sig* is sent to any process with effective group ID *id* .

If *idtype* is P_CID , *sig* is sent to any process with scheduler class ID *id* (see priocntl(2)).

If *idtype* is P_ALL , *sig* is sent to all processes and *id* is ignored.

If *id* is P_MYID , the value of *id* is taken from the calling process.

The process with a process ID of 0 is always excluded. The process with a process ID of 1 is excluded unless *idtype* is equal to P_PID .

The sigsendset( ) function provides an alternate interface for sending signals to sets of processes. This function sends signals to the set of processes specified by *psp* . *psp* is a pointer to a structure of type procset_t , defined in <sys/procset.h> , which includes the following members:

```
idop_t    p_op;
idtype_t  p_lidtype;
id_t      p_lid;
idtype_t  p_ridtype;
id_t      p_rid;
```

The p_lidtype and p_lid members specify the ID type and ID of one (“left”) set of processes; the p_ridtype and p_rid members specify the ID type

and ID of a second (“right”) set of processes. ID types and ID s are specified just as for the *idtype* and *id* arguments to `sigsend( )` . The `p_op` member specifies the operation to be performed on the two sets of processes to get the set of processes the function is to apply to. The valid values for `p_op` and the processes they specify are:

`POP_DIFF`	Set difference: processes in left set and not in right set.
`POP_AND`	Set intersection: processes in both left and right sets.
`POP_OR`	Set union: processes in either left or right set or both.
`POP_XOR`	Set exclusive-or: processes in left or right set but not in both.

RETURN VALUES

Upon successful completion, `0` is returned. Otherwise, `–1` is returned and `errno` is set to indicate the error.

ERRORS

The `sigsend( )` and `sigsendset( )` functions will fail if:

`EINVAL`	The *sig* argument is not a valid signal number, or the *idtype* argument is not a valid idtype field.
`EINVAL`	The *sig* argument is `SIGKILL,` *idtype* is `P_PID` and *id* is `1` (proc1).
`EPERM`	The 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 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.

The `sigsendset( )` function will fail if:

`EFAULT`	The *psp* argument 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

Trusted Solaris 8 Reference Manual

`getpid`(2) , `kill`(2) , `priocntl`(2)

SunOS 5.8 Reference Manual

`kill`(1) , `signal`(3C) , `signal`(3HEAD)

NAME

stat, lstat, fstat – Get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char * path, struct stat * buf);

int lstat(const char * path, struct stat * buf);

int fstat(int fildes, struct stat * buf);
```

DESCRIPTION

The `stat()` function obtains information about the file pointed to by *path* . Read, write, or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

The `lstat()` function obtains file attributes similar to `stat()` , except when the named file is a symbolic link; in that case `lstat()` returns information about the link, while `stat()` returns information about the file the link references.

The `fstat()` function obtains information about an open file known by the file descriptor *fildes* , obtained from a successful `open`(2) , `creat`(2) , `dup`(2) , `fcntl`(2) , or `pipe`(2) function.

The *buf* argument is a pointer to a `stat` structure into which information is placed concerning the file. A `stat` structure includes the following members:

```
mode_t   st_mode;     /* File mode (see mknod(2)) */
ino_t    st_ino;      /* Inode number */
dev_t    st_dev;      /* ID of device containing */
                      /* a directory entry for this file */
dev_t    st_rdev;     /* ID of device */
                      /* This entry is defined only for */
                      /* char special or block special files */
nlink_t  st_nlink;    /* Number of links */
uid_t    st_uid;      /* User ID of the file's owner */
gid_t    st_gid;      /* Group ID of the file's group */
off_t    st_size;     /* File size in bytes */
time_t   st_atime;    /* Time of last access */
time_t   st_mtime;    /* Time of last data modification */
time_t   st_ctime;    /* Time of last file status change */
                      /* Times measured in seconds since */
                      /* 00:00:00 UTC, Jan. 1, 1970 */
long     st_blksize;  /* Preferred I/O block size */
blkcnt_t st_blocks;   /* Number of 512 byte blocks allocated*/
```

Descriptions of structure members are as follows:

`st_mode` The mode of the file as described in `mknod`(2) . In addition to the modes described in `mknod()` , the mode of a file may also be `S_IFLNK` if the file is a symbolic link. `S_IFLNK` may only be returned by `lstat()` .

st_ino	This field uniquely identifies the file in a given file system. The pair st_ino and st_dev uniquely identifies regular files.
st_dev	This field uniquely identifies the file system that contains the file. Its value may be used as input to the ustat( ) function to determine more information about this file system. No other meaning is associated with this value.
st_rdev	This field should be used only by administrative commands. It is valid only for block special or character special files and only has meaning on the system where the file was configured.
st_nlink	This field should be used only by administrative commands.
st_uid	The user ID of the file's owner.
st_gid	The group ID of the file's group.
st_size	For regular files, this is the address of the end of the file. For block special or character special, this is not defined. See also pipe(2) .
st_atime	Time when file data was last accessed. Changed by the following functions: creat( ) , mknod( ) , pipe( ) , utime(2) , and read(2) .
st_mtime	Time when data was last modified. Changed by the following functions: creat( ) , mknod( ) , pipe( ) , utime( ) , and write(2) .
st_ctime	Time when file status was last changed. Changed by the following functions: chmod( ) , chown( ) , creat( ) , link(2) , mknod( ) , pipe( ) , unlink(2) , 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.

stat( ) , lstat( ) , and fstat( ) require mandatory read access to the final component of *path* . If the file descriptor is open only for writing, fstat( ) requires mandatory read access to the object to which the descriptor refers. To override these restrictions, the calling process may assert the PRIV_FILE_MAC_READ privilege in its set of effective privileges.

If the calling process does not have mandatory read access, `stat()`, `lstat()`, and `fstat()` return fixed values for some elements of the `stat` structure.

RETURN VALUES

Upon successful completion, `0` is returned. Otherwise, `–1` is returned and `errno` is set to indicate the error.

ERRORS

The `stat()`, `fstat()`, and `lstat()` functions will fail if:

`EOVERFLOW` The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by *buf*.

The `stat()` and `lstat()` functions will fail if:

`EACCES` Search permission is denied for a component of the path prefix. 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` The *buf* or *path* argument points to an illegal address.

`EINTR` A signal was caught during the execution of the `stat()` or `lstat()` function.

`ELOOP` Too many symbolic links were encountered in translating *path*.

`ENAMETOOLONG` The length of the *path* argument exceeds `PATH_MAX`, or the length of a *path* component exceeds `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.

`ENOENT` The named file does not exist or is the null pathname.

`ENOLINK` The *path* argument points to a remote machine and the link to that machine is no longer active.

`ENOTDIR` A component of the path prefix is not a directory.

`EOVERFLOW` A component is too large to store in the structure pointed to by *buf*.

The `fstat()` function will fail if:

`EBADF` The *fildes* argument is not a valid open file descriptor.

`EFAULT` The *buf* argument points to an illegal address.

`EINTR` A signal was caught during the execution of the `fstat()` function.

`ENOLINK` The *fildes* argument points to a remote machine and the link to that machine is no longer active.

EOVERFLOW A component is too large to store in the structure pointed to by *buf* .

USAGE

The stat() , fstat() , and lstat() functions have transitional interfaces for 64-bit file offsets. See lf64(5) .

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	stat() and fstat() are Async-Signal-Safe

SUMMARY OF TRUSTED SOLARIS CHANGES

stat() , lstat() , and fstat() require mandatory read access to the final component of *path* . If the file descriptor is open only for writing, fstat() requires mandatory read access to the object to which the descriptor refers. To override these restrictions, the calling process may assert the PRIV_FILE_MAC_READ privilege in its set of effective privileges.

To override access restrictions, the calling process of stat() or lstat() may also assert one or both of these privileges: PRIV_FILE_DAC_SEARCH and PRIV_FILE_MAC_SEARCH .

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

Trusted Solaris 8 Reference Manual

chmod(2) , chown(2) , creat(2) , fcntl(2) , link(2) , mknod(2) , open(2) , read(2) , unlink(2) , write(2)

SunOS 5.8 Reference Manual

dup(2) , pipe(2) , time(2) , utime(2) , fattach(3C) , stat(3HEAD) , attributes(5)

NOTES

If you use chmod(2) to change the file group owner permissions on a file with ACL entries, both the file group owner permissions and the ACL mask are changed to the new permissions. Be aware that the new ACL mask permissions may change the effective permissions for additional users and groups who have ACL entries on the file.

NAME

statvfs, fstatvfs – Get file system information

SYNOPSIS

```
#include <sys/types.h>
#include <sys/statvfs.h>
int statvfs(const char * path, struct statvfs * buf);

int fstatvfs(int fildes, struct statvfs * buf);
```

DESCRIPTION

The `statvfs( )` function returns a "generic superblock" describing a file system; it can be used to acquire information about mounted file systems. The *buf* argument is a pointer to a structure (described below) that is filled by the function.

The *path* argument should name a file that resides on that file system. The file system type is known to the operating system. Read, write, or execute permission for the named file is not required, but all directories listed in the path name leading to the file must be searchable.

The `statvfs` structure pointed to by *buf* includes the following members:

```
u_long      f_bsize;             /* preferred file system block size */
u_long      f_frsize;            /* fundamental filesystem block
                                    (size if supported) */
fsblkcnt_t  f_blocks;            /* total # of blocks on file system
                                    in units of f_frsize */
fsblkcnt_t  f_bfree;             /* total # of free blocks */
fsblkcnt_t  f_bavail;            /* # of free blocks avail to
                                    non-super-user */
fsfilcnt_t  f_files;             /* total # of file nodes (inodes) */
fsfilcnt_t  f_ffree;             /* total # of free file nodes */
fsfilcnt_t  f_favail;            /* # of inodes avail to
                                    non-super-user*/
u_long      f_fsid;              /* file system id (dev for now) */
char        f_basetype[FSTYPSZ]; /* target fs type name,
                                    null-terminated */
u_long      f_flag;              /* bit mask of flags */
u_long      f_namemax;           /* maximum file name length */
char        f_fstr[32];          /* file system specific string */
u_long      f_filler[16];        /* reserved for future expansion */
```

The `f_basetype` member contains a null-terminated FSType name of the mounted target.

The following values can be returned in the `f_flag` field:

```
ST_RDONLY    0x01    /* read-only file system */
ST_NOSUID    0x02    /* does not support setuid/setgid semantics */
ST_NOTRUNC   0x04    /* does not truncate file names longer than
                        NAME_MAX */
```

The `fstatvfs( )` function is similar to `statvfs( )`, except that the file named by *path* in `statvfs( )` is instead identified by an open file descriptor *fildes* obtained from a successful `open`(2), `creat`(2), `dup`(2), `fcntl`(2), or `pipe`(2) function call.

RETURN VALUES

`statvfs( )` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `statvfs( )` and `fstatvfs( )` functions will fail if:

`EOVERFLOW` One of the values to be returned cannot be represented correctly in the structure pointed to by *buf*.

The `statvfs( )` function will fail if:

`EACCES` Search permission is denied on a component of the path prefix. 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` The *path* or *buf* argument points to an illegal address.

`EINTR` A signal was caught during the execution of the `statvfs( )` function.

`EIO` An I/O error occurred while reading the file system.

`ELOOP` Too many symbolic links were encountered in translating *path*.

`ENAMETOOLONG` The length of a *path* component exceeds `NAME_MAX` characters, or the length of *path* The exceeds `PATH_MAX` characters.

`ENOENT` Either a component of the path prefix or the file referred to by *path* does not exist.

`ENOLINK` The *path* argument points to a remote machine and the link to that machine is no longer active.

ENOTDIR	A component of the path prefix of *path* is not a directory.

The `fstatvfs()` function will fail if:

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	The *fildes* argument is not an open file descriptor.
EFAULT	The *buf* argument points to an illegal address.
EINTR	A signal was caught during the execution of the `fstatvfs()` function.
EIO	An I/O error occurred while reading the file system.

USAGE

The `statvfs()` and `fstatvfs()` functions have transitional interfaces for 64-bit file offsets. See `lf64`(5) .

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

SEE ALSO

Trusted Solaris 8 Reference Manual

`chmod`(2) , `chown`(2) , `creat`(2) , `fcntl`(2) , `link`(2) , `mknod`(2) , `open`(2) , `read`(2) , `unlink`(2) , `write`(2)

SunOS 5.8 Reference Manual

`dup`(2) , `pipe`(2) , `time`(2) , `utime`(2)

BUGS

The values returned for `f_files` , `f_ffree` , and `f_favail` may not be valid for NFS mounted file systems.

NAME

stime – Set system time and date

SYNOPSIS

```
#include <unistd.h>
int stime(const time_t *tp);
```

DESCRIPTION

The `stime()` function sets the system's idea of the time and date. The *tp* argument points to the value of time as measured in seconds from 00:00:00 UTC January 1, 1970. The calling process must have the `PRIV_SYS_CONFIG` privilege in order to use this system call.

RETURN VALUES

`stime()` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `stime()` function will fail if:

`EINVAL` The *tp* argument points to an invalid (negative) time value.

`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

SunOS 5.8 Reference Manual

`time`(2)

NAME

swapctl – Manage swap space

SYNOPSIS

```
#include <sys/stat.h>
#include <sys/swap.h>
int swapctl(int cmd, void *arg);
```

DESCRIPTION

The `swapctl( )` function adds, deletes, or returns information about swap resources. *cmd* specifies one of the following options contained in `<sys/swap.h>`:

```
SC_ADD        /* add a resource for swapping */
SC_LIST       /* list the resources for swapping */
SC_REMOVE     /* remove a resource for swapping */
SC_GETNSWP    /* return number of swap resources */
```

When `SC_ADD` or `SC_REMOVE` is specified, *arg* is a pointer to a `swapres` structure containing the following members:

```
char    *sr_name;    /* pathname of resource */
off_t   sr_start;    /* offset to start of swap area */
off_t   sr_length;   /* length of swap area */
```

The `sr_start` and `sr_length` members are specified in 512-byte blocks. A swap resource can only be removed by specifying the same values for the `sr_start` and `sr_length` members as were specified when it was added. Swap resources need not be removed in the order in which they were added.

When `SC_LIST` is specified, *arg* is a pointer to a `swaptable` structure containing the following members:

```
int             swt_n;       /* number of swapents following */
struct swapent  swt_ent[];   /* array of swt_n swapents */
```

A `swapent` structure contains the following members:

```
char   *ste_path;    /* name of the swap file */
off_t  ste_start;    /* starting block for swapping */
off_t  ste_length;   /* length of swap area */
long   ste_pages;    /* number of pages for swapping */
long   ste_free;     /* number of ste_pages free */
long   ste_flags;    /* ST_INDEL bit set if swap file */
                     /* is now being deleted */
```

The SC_LIST function causes swapctl( ) to return at most swt_n entries. The return value of swapctl( ) is the number actually returned. The ST_INDEL bit is turned on in ste_flags if the swap file is in the process of being deleted.

When SC_GETNSWP is specified, swapctl( ) returns as its value the number of swap resources in use. *arg* is ignored for this operation.

The SC_ADD and SC_REMOVE functions will fail if calling process does not have appropriate privileges.

RETURN VALUES

Upon successful completion, 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, swapctl( ) returns a value of –1 and sets errno to indicate an error.

ERRORS

Under the following conditions, the function swapctl( ) fails and sets errno to:

EEXIST	Part of the range specified by sr_start and sr_length is already being used for swapping on the specified resource (SC_ADD).
EFAULT	Either *arg*, sr_name, or ste_path points to an illegal address.
EINVAL	The specified function value is not valid, the path specified is not a swap resource (SC_REMOVE), part of the range specified by sr_start and sr_length lies outside the resource specified (SC_ADD), or the specified swap area is less than one page (SC_ADD).
EISDIR	The path specified for SC_ADD is a directory.
ELOOP	Too many symbolic links were encountered in translating the pathname provided to SC_ADD or SC_REMOVE.
ENAMETOOLONG	The length of a component of the path specified for SC_ADD or SC_REMOVE exceeds NAME_MAX characters or the length of the path exceeds PATH_MAX characters and _POSIX_NO_TRUNC is in effect.
ENOENT	The pathname specified for SC_ADD or SC_REMOVE does not exist.
ENOMEM	An insufficient number of struct swapent structures were provided to SC_LIST, or there were insufficient system storage resources available during an SC_ADD or SC_REMOVE, or

	the system would not have enough swap space after an SC_REMOVE.
ENOSYS	The pathname specified for SC_ADD or SC_REMOVE is not a file or block special device.
ENOTDIR	Pathname provided to SC_ADD or SC_REMOVE contained a component in the path prefix that was not a directory.
EPERM	The 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.

Additionally, the swapctl( ) function will fail for 32-bit interfaces if:

EOVERFLOW	The amount of swap space configured on the machine is too large to be represented by a 32-bit quantity.

EXAMPLES

EXAMPLE 1 The usage of the SC_GETNSWP and SC_LIST commands.

The following example demonstrates the usage of the SC_GETNSWP and SC_LIST commands.

```
#include <sys/stat.h>
#include <sys/swap.h>
#include <stdio.h>

#define MAXSTRSIZE 80

main(argc, argv)
    int             argc;
    char            *argv[];
{
    swaptbl_t       *s;
    int             i, n, num;
    char            *strtab;    /* string table for path names */

again:
    if ((num = swapctl(SC_GETNSWP, 0)) == -1) {
        perror("swapctl: GETNSWP");
        exit(1);
    }
    if (num == 0) {
        fprintf(stderr, "No Swap Devices Configured\n");
        exit(2);
    }
    /* allocate swaptable for num+1 entries */
    if ((s = (swaptbl_t *)
```

```
            malloc(num * sizeof(swapent_t) +
                sizeof(struct swaptable))) ==
            (void *) 0) {
            fprintf(stderr, "Malloc Failed\n");
            exit(3);
        }
        /* allocate num+1 string holders */
        if ((strtab = (char *)
            malloc((num + 1) * MAXSTRSIZE)) == (void *) 0) {
            fprintf(stderr, "Malloc Failed\n");
            exit(3);
        }
        /* initialize string pointers */
        for (i = 0; i < (num + 1); i++) {
            s->swt_ent[i].ste_path = strtab + (i * MAXSTRSIZE);
        }

        s->swt_n = num + 1;
        if ((n = swapctl(SC_LIST, s)) < 0) {
            perror("swapctl");
            exit(1);
        }
        if (n > num) {        /* more were added */
            free(s);
            free(strtab);
            goto again;
        }
        for (i = 0; i < n; i++)
            printf("%s %ld\n",
                s->swt_ent[i].ste_path, s->swt_ent[i].ste_pages);
    }
```

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

The `symlink()` function creates a symbolic link *name2* to the file *name1*. Either name may be an arbitrary pathname, the files need not be on the same file system, and *name1* may be nonexistent.

The file to which the symbolic link points is used when an `open`(2) operation is performed on the link. A `stat()` operation performed on a symbolic link returns the linked-to file, while an `lstat()` operation returns information about the link itself. See `stat`(2). Unexpected results may occur when a symbolic link is made to a directory. To avoid confusion in applications, the `readlink`(2) call can be used to read the contents of a symbolic link.

The containing directory cannot be a multilevel directory. There is no privilege to bypass this restriction.

The link is created with its sensitivity label set to the sensitivity label of the calling process, and its user ID set to the effective user ID of the calling process. If the file system was not mounted with the BSD file-creation semantics flag and the set-gid bit of the parent directory is clear, the new link's group ID is set to the group ID of the directory in which the link is created. The new link's permission bits are set to 0777. Even when the containing directory has a default access control list (ACL), no ACL is set on the new link.

RETURN VALUES

`symlink()` returns:

`0` On success.

`–1` On failure, sets `errno` to indicate the error, and the symbolic link is not made.

ERRORS

`EACCES` Search permission is denied for a component of the path prefix of *name2*. 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 containing directory of *name2*. 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 where the entry for the new symbolic link is being placed cannot be extended

	because the user's quota of disk blocks on that file system has been exhausted; the new symbolic link cannot be created because the user's quota of disk blocks on that file system has been exhausted; or the user's quota of inodes on the file system where the file is being created has been exhausted.
EEXIST	The file referred to by *name2* already exists.
EFAULT	The *name1* or *name2* argument points to an illegal address.
EIO	An I/O error occurs while reading from or writing to the file system.
ELOOP	Too many symbolic links are encountered in translating *name2*.
ENAMETOOLONG	The length of the *name2* argument exceeds PATH_MAX, or the length of a *name2* component exceeds NAME_MAX while _POSIX_NO_TRUNC is in effect.
ENOENT	A component of the path prefix of *name2* does not exist.
ENOSPC	The directory in which the entry for the new symbolic link is being placed cannot be extended because no space is left on the file system containing the directory; the new symbolic link cannot be created because no space is left on the file system which will contain the link; or there are no free inodes on the file system on which the file is being created.
ENOSYS	The file system does not support symbolic links
ENOTDIR	A component of the path prefix of *name2* is not a directory.
EROFS	The file *name2* would reside on a read-only file system.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

The containing directory cannot be a multilevel directory.

SEE ALSO

Trusted Solaris 8 Reference Manual

`link`(2), `open`(2), `readlink`(2), `stat`(2), `unlink`(2)

SunOS 5.8 Reference Manual

`cp`(1)

NAME

sysinfo – Get and set system information strings

SYNOPSIS

```
#include <sys/systeminfo.h>
long sysinfo(int command, char *buf, long count);
```

DESCRIPTION

The `sysinfo( )` function copies information relating to the operating system on which the process is executing into the buffer pointed to by *buf*. It can also set certain information where appropriate *commands* are available. The *count* parameter indicates the size of the buffer.

The POSIX P1003.1 interface (see `standards`(5)) `sysconf`(3C) provides a similar class of configuration information, but returns an integer rather than a string.

The values for *command* are as follows:

`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, sun4u, sun4d, or sun4m.
SI_ARCHITECTURE	Copy into the array pointed to by *buf* a string describing the basic instruction set architecture of the current system, for example, *sparc*, mc68030, m32100, or *i386*. These names may not match predefined names in the C language compilation system.
SI_ISALIST	Copy into the array pointed to by *buf* the names of the variant instruction set architectures executable on the current system. The names are space-separated and are ordered in the sense of best performance. That is, earlier-named instruction sets may contain more instructions than later-named instruction sets; a program that is compiled for an earlier-named instruction set will most likely run faster on this machine than the same program compiled for a later-named instruction set. Programs compiled for an instruction set that does not appear in the list will most likely experience performance degradation or not run at all on this machine. The instruction set names known to the system are listed in isalist(5); these names may or may not match predefined names or compiler options 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 text 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.
SI_DHCP_CACHE	Copy into the array pointed to by *buf* an ASCII string consisting of the ASCII hexidecimal encoding of the name of the interface configured by boot(1M) followed by the DHCPACK reply from the server. This command is intended for use only by the dhcpagent(1M) DHCP client daemon for the purpose of adopting the DHCP maintenance of the interface configured by boot.

RETURN VALUES

Upon successful completion, the value returned indicates the buffer size in bytes required to hold the complete value and the terminating null character. If this value is no greater than the value passed in *count*, the entire string was copied. If this value is greater than *count*, the string copied into *buf* has been truncated to *count –1* bytes plus a terminating null character.

Otherwise, –1 is returned and errno is set to indicate the error.

ERRORS

The sysinfo( ) function will fail if:

EFAULT	The *buf* argument does not point to a valid address.
EINVAL	The 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 estimation 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 SI_SET_HOSTNAME, and SI_SET_SRPC_DOMAIN operations.

SEE ALSO

SunOS 5.8 Reference Manual

boot(1M), dhcpagent(1M), uname(2), gethostid(3C), gethostname(3C), sysconf(3C), isalist(5), standards(5)

NAME

tokmapper – Manipulate kernel token mapping caches

SYNOPSIS

cc [*flags*…] *file*… –ltsol

```
#include <netinet/in.h>
#include <sys/tiuser.h>
 #include <sys/tsol/tndb.h>
int tokmapper(int cmd, void *buf);
```

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.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWtsu

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

Trusted Solaris 8 Reference Manual

`tokmapd`(1M), `tokmapctl`(1M)

SunOS 5.8 Reference Manual

`attributes`(5)

NAME

uadmin – Administrative control

SYNOPSIS

```
#include <sys/uadmin.h>
int uadmin(int cmd, int fcn, uintptr_t mdep);
```

DESCRIPTION

The `uadmin( )` function provides control for basic administrative functions. This function is tightly coupled to the system administrative procedures and is not intended for general use. The argument *mdep* is provided for machine-dependent use and is not defined here.

As specified by *cmd*, the following commands are available:

`A_SHUTDOWN` The system is shut down. All user processes are killed, the buffer cache is flushed, and the root file system is unmounted. The action to be taken after the system has been shut down is specified by *fcn*. The functions are generic; the hardware capabilities vary on specific machines.

- `AD_HALT` Halt the processor(s).
- `AD_POWEROFF` Halt the processor(s) and turn off the power.
- `AD_BOOT` Reboot the system, using the kernel file.
- `AD_IBOOT` Interactive reboot; user is prompted for bootable program name.

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.
A_DUMP	The system is forced to panic immediately without any further processing and a crash dump is written to the dump device (see dumpadm(1M)). The action to be taken next is specified by *fcn* as above.
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.

Otherwise, −1 is returned and errno is set to indicate the error.

ERRORS

The uadmin( ) function will fail if:

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 commands. The calling process must have the PRIV_SYS_MOUNT privilege in order to perform the A_REMOUNT command.

SEE ALSO

Trusted Solaris 8 Reference Manual `uadmin`(1M)

SunOS 5.8 Reference Manual `kernel`(1M)

NAME

ulimit – Get and set process limits

SYNOPSIS

```
#include <ulimit.h>
long ulimit(int cmd, /* newlimit */...);
```

DESCRIPTION

The `ulimit()` function provides for control over process limits. It is effective in limiting the growth of regular files. Pipes are limited to `PIPE_MAX` bytes.

The *cmd* values, defined in `<ulimit.h>`, include:

`UL_GETFSIZE` Return the soft file size limit of the process. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read. The return value is the integer part of the soft file size limit divided by 512. If the result cannot be represented as a `long int`, the result is unspecified.

`UL_SETFSIZE` Set the hard and soft file size limits for output operations of the process to the value of the second argument, taken as a `long int`. Any process may decrease its own hard limit, but only a process with an effective `PRIV_SYS_CONFIG` privilege may increase the limit. The new file size limit is returned. The hard and soft file size limits are set to the specified value multiplied by 512. If the result would overflow an `rlimit_t`, the actual value set is unspecified.

`UL_GMEMLIM` Get the maximum possible break value (see `brk`(2)).

`UL_GDESLIM` Get the current value of the maximum number of open files per process configured in the system.

RETURN VALUES

Upon successful completion, `ulimit()` returns the value of the requested limit. Otherwise, `–1` is returned, the limit is not changed, and `errno` is set to indicate the error.

ERRORS

The `ulimit()` function will fail if:

`EINVAL` The *cmd* argument is not valid.

`EPERM` A process not having an effective `PRIV_SYS_CONFIG` privilege attempts to increase its file size limit.

USAGE

Since all return values are permissible in a successful situation, an application wishing to check for error situations should set `errno` to `0`, then call `ulimit()`, and if it returns `–1`, check if `errno` is non-zero.

The `getrlimit()` and `setrlimit()` functions provide a more general interface for controlling process limits, and are preferred over `ulimit()`. See `getrlimit`(2).

SUMMARY OF TRUSTED SOLARIS CHANGES

The PRIV_SYS_CONFIG privilege is checked.

SEE ALSO

Trusted Solaris 8 Reference Manual

getrlimit(2), write(2)

SunOS 5.8 Reference Manual

brk(2)

NAME

umount, umount2 – Unmount a file system

SYNOPSIS

```
#include <sys/mount.h>
int umount(const char * file);

int umount2(const char * file, int mflag);
```

DESCRIPTION

The `umount( )` function requests that a previously mounted file system contained on the block special device or directory identified by *file* be unmounted. The *file* argument is a pointer to a path name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

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

`umount( )` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `umount( )` function will fail if:

`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`	The file pointed to by *file* points to an illegal address.
`EINVAL`	The file pointed to by *file* is not mounted.
`ENOENT`	The file pointed to by *file* does not exist.
`ELOOP`	Too many symbolic links were encountered in translating the path pointed to by *file* .
`ENAMETOOLONG`	The length of the *file* argument exceeds `PATH_MAX` , or the length of a *file* component exceeds `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.

ENOLINK	The file pointed to by *file* is on a remote machine, and the link to that machine is no longer active.
ENOTBLK	The file pointed to by *file* is not a block special device.
EPERM	The 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 .
	file is not a file system of type *namefs* and the calling process has not asserted the PRIV_SYS_MOUNT privilege.
EREMOTE	The file pointed to by *file* is remote.

USAGE

The umount( ) function may be invoked by a calling process with the appropriate privilege.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access or ownership checks.

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
Trusted Solaris 8 Reference Manual

mount(2)

NAME

umount, umount2 – Unmount a file system

SYNOPSIS

```
#include <sys/mount.h>
int umount(const char * file);

int umount2(const char * file, int mflag);
```

DESCRIPTION

The `umount()` function requests that a previously mounted file system contained on the block special device or directory identified by *file* be unmounted. The *file* argument is a pointer to a path name. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

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

`umount()` returns:

`0` On success.

`–1` On failure, and sets `errno` to indicate the error.

ERRORS

The `umount()` function will fail if:

`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`	The file pointed to by *file* points to an illegal address.
`EINVAL`	The file pointed to by *file* is not mounted.
`ENOENT`	The file pointed to by *file* does not exist.
`ELOOP`	Too many symbolic links were encountered in translating the path pointed to by *file* .
`ENAMETOOLONG`	The length of the *file* argument exceeds `PATH_MAX` , or the length of a *file* component exceeds `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.

ENOLINK	The file pointed to by *file* is on a remote machine, and the link to that machine is no longer active.
ENOTBLK	The file pointed to by *file* is not a block special device.
EPERM	The 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 .
	file is not a file system of type *namefs* and the calling process has not asserted the PRIV_SYS_MOUNT privilege.
EREMOTE	The file pointed to by *file* is remote.

USAGE

The umount( ) function may be invoked by a calling process with the appropriate privilege.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access or ownership checks.

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
Trusted Solaris 8 Reference Manual

mount(2)

NAME

unlink – Remove directory entry

SYNOPSIS

```
#include <unistd.h>
int unlink(const char *path);
```

DESCRIPTION

The `unlink()` function removes a link to a file. If *path* names a symbolic link, `unlink()` removes the symbolic link named by *path* and does not affect any file or directory named by the contents of the symbolic link. Otherwise, `unlink()` removes the link named by the pathname pointed to by *path* and decrements the link count of the file referenced by the link.

When the file's link count becomes 0 and no process has the file open, the space occupied by the file will be freed and the file will no longer be accessible. If one or more processes have the file open when the last link is removed, the link will be removed before `unlink()` returns, but the removal of the file contents will be postponed until all references to the file are closed.

The *path* argument must not name a directory unless the process has asserted the `PRIV_SYS_CONFIG` privilege and the implementation supports using `unlink()` on directories.

Upon successful completion, `unlink()` will mark for update the `st_ctime` and `st_mtime` fields of the parent directory. If the file's link count is not 0, the `st_ctime` field of the file will be marked for update.

RETURN VALUES

Upon successful completion, `0` is returned. Otherwise, `–1` is returned, `errno` is set to indicate the error, and the file is not unlinked.

ERRORS

The `unlink()` function will fail if:

`EACCES` Search permission is denied for a component of the *path* prefix.

Write permission is denied on the directory containing the link to be removed. 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 parent directory has the sticky bit set and the file is not writable by the user; or the user does not own the parent directory and the user does not own the file. 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	The *path* argument points to an illegal address.
EINTR	A signal was caught during the execution of the `unlink()` function.
ELOOP	Too many symbolic links were encountered in translating *path*.
ENAMETOOLONG	The length of the *path* argument exceeds `PATH_MAX`, or the length of a *path* component exceeds `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.
ENOENT	The named file does not exist or is a null pathname.
ENOLINK	The *path* argument points to a remote machine and the link to that machine is no longer active.
ENOTDIR	A component of the *path* prefix is not a directory.
EPERM	The named file is a directory and 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.

The `unlink()` function may fail if:

ENAMETOOLONG	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.
ETXTBSY	The entry to be unlinked is the last directory entry to a pure procedure (shared text) file that is being executed.

USAGE

Applications should use `rmdir`(2) to remove a directory.

ATTRIBUTES

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Async-Signal-Safe

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.

SEE ALSO

Trusted Solaris 8 Reference Manual

`rm`(1), `link`(2), `open`(2), `rmdir`(2)

SunOS 5.8 Reference Manual

`close`(2), `remove`(3C), `attributes`(5)

NAME

utimes – Set file access and modification times

SYNOPSIS

```
#include <sys/time.h>
int utimes(const char *path, const struct timeval times[2]);
```

DESCRIPTION

The `utimes( )` function sets the access and modification times of the file pointed to by the *path* argument to the value of the *times* argument. It allows time specifications accurate to the microsecond.

The *times* argument is an array of `timeval` structures. The first array member represents the date and time of last access, and the second member represents the date and time of last modification. The times in the `timeval` structure are measured in seconds and microseconds since the Epoch, although rounding toward the nearest second may occur.

If the *times* argument is a null pointer, the access and modification times of the file are set to the current time. A process must be the owner of the file or must assert the `PRIV_FILE_OWNER` privilege to use this call in this manner. Upon completion, `utimes( )` will mark the time of the last file status change, `st_ctime`, for update.

RETURN VALUES

`utimes( )` returns:

`0` On success.

`–1` On failure, sets `errno` to indicate the error, and the file times will not be affected

ERRORS

The `utimes( )` function will fail if:

`EACCES` Search permission is denied by a component of the path prefix. To override the calling process may assert one or both of these privileges: `PRIV_FILE_DAC_SEARCH` and `PRIV_FILE_MAC_SEARCH`.

The *times* argument is a null pointer and the effective user ID of the process does not match the owner of the file and write access is denied. 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` The *path* or *times* argument points to an illegal address.

`EINTR` A signal was caught during the execution of the `utimes( )` function.

EINVAL	The number of microseconds specified in one or both of the `timeval` structures pointed to by *times* was greater than or equal to 1,000,000 or less than 0.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in resolving *path*.
ENAMETOOLONG	The length of the *path* argument exceeds PATH_MAX or a pathname component is longer than NAME_MAX.
ENOLINK	The *path* argument points to a remote machine and the link to that machine is no longer active.
ENOENT	A component of *path* does not name an existing file or *path* is an empty string.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	The *times* argument is not a null pointer and the calling process's effective user ID has write access to the file but does not match the owner of the file and the calling process does not have the appropriate privileges.To override this restriction, the calling process may assert the PRIV_FILE_OWNER privilege.
EROFS	The file system containing the file is read-only.

The `utimes()` function may fail if:

ENAMETOOLONG	Path name resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.

SUMMARY OF TRUSTED SOLARIS CHANGES

Appropriate privilege is required to override access checks.

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

Trusted Solaris 8 Reference Manual

`stat`(2)

NAME

vfork – Spawn new process in a virtual memory efficient way

SYNOPSIS

```
#include <unistd.h>
pid_t vfork(void);
```

DESCRIPTION

The `vfork( )` function creates new processes without fully copying the address space of the old process. This function is useful in instances where the purpose of a `fork`(2) operation would be to create a new system context for an `execve( )` operation (see `exec`(2)).

Unlike with the `fork( )` function, the child process borrows the parent's memory and thread of control until a call to `execve( )` or an exit (either abnormally or by a call to `_exit( )` (see `exit`(2)). The parent process is suspended while the child is using its resources.

In a multithreaded application, `vfork( )` borrows only the thread of control that called `vfork( )` in the parent; that is, the child contains only one thread. In that sense, `vfork( )` behaves like `fork( )`.

The `vfork( )` function can normally be used the same way as `fork( )`. The procedure that called `vfork( )`, however, should not return while running in the child's context, since the eventual return from `vfork( )` would be to a stack frame that no longer exists. The `_exit( )` function should be used in favor of `exit`(3C) if unable to perform an `execve( )` operation, since `exit( )` will flush and close standard I/O channels, and thereby corrupt the parent process's standard I/O data structures. The `_exit( )` function should be used even with `fork( )` to avoid flushing the buffered data twice.

RETURN VALUES

Upon successful completion, `vfork( )` returns `0` to the child process and returns the process ID of the child process to the parent process. Otherwise, `–1` is returned to the parent process, no child process is created, and `errno` is set to indicate the error.

ERRORS

The `vfork( )` function will fail if:

`EAGAIN` The system-imposed limit on the total number of processes under execution (either system-quality or by a single user) would be exceeded. 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

A process with the `PRIV_SYS_MAXPROC` privilege may override the limit on the number of processes a user may have.

SEE ALSO

Trusted Solaris 8 Reference Manual

`exec`(2), `fork`(2)

SunOS 5.8 Reference Manual

`exit`(2), `ioctl`(2), `wait`(2), `exit`(3C)

NOTES

The use of `vfork( )` for any purpose other than as a prelude to an immediate call to a function from the `exec` family or to `_exit( )` is not advised.

The `vfork( )` function is unsafe in multithreaded applications.

This function will be eliminated in a future release. The memory sharing semantics of `vfork( )` can be obtained through other mechanisms.

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, pwrite, writev, writel, pwritel, writevl – Write on a file

SYNOPSIS

```
#include <unistd.h>
ssize_t write(int fildes, const void * buf, size_t nbyte);

ssize_t pwrite(int fildes, const void * buf, size_t nbyte, off_t offset);

#include <sys/uio.h>
ssize_t writev(int fildes, const struct iovec * iov, int iovcnt);

#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);
```

DESCRIPTION

The `write()` function attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor, *fildes* .

If *nbyte* is `0` , `write()` will return `0` and have no other results if the file is a regular file; otherwise, the results are unspecified.

On a regular file or other file capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file offset associated with *fildes* . Before successful return from `write()` , the file offset is incremented by the number of bytes actually written. On a regular file, if this incremented file offset is greater than the length of the file, the length of the file will be set to this file offset.

If the `O_SYNC` flag of the file status flags is set and *fildes* refers to a regular file, a successful `write()` does not return until the data is delivered to the underlying hardware.

If *fildes* refers to a socket, `write()` is equivalent to `send`(3SOCKET) with no flags set.

On a file not capable of seeking, writing always takes place starting at the current position. The value of a file offset associated with such a device is undefined.

If the `O_APPEND` flag of the file status flags is set, the file offset will be set to the end of the file prior to each write and no intervening file modification operation will occur between changing the file offset and the write operation.

For regular files, no data transfer will occur past the offset maximum established in the open file description with *fildes* .

A `write()` to a regular file is blocked if mandatory file / record locking is set (see `chmod`(2)), and there is a record lock owned by another process on the segment of the file to be written:

- If O_NDELAY or O_NONBLOCK is set, write( ) returns –1 and sets errno to EAGAIN .
- If O_NDELAY and O_NONBLOCK are clear, write( ) sleeps until all blocking locks are removed or the write( ) is terminated by a signal.

If a write( ) requests that more bytes be written than there is room for—for example, if the write would exceed the process file size limit (see getrlimit(2) and ulimit(2)), the system file size limit, or the free space on the device—only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write( ) of 512-bytes returns 20 . The next write( ) of a non-zero number of bytes gives a failure return (except as noted for pipes and FIFO below).

If write( ) is interrupted by a signal before it writes any data, it will return –1 with errno set to EINTR .

If write( ) is interrupted by a signal after it successfully writes some data, it will return the number of bytes written.

If the value of *nbyte* is greater than SSIZE_MAX , the result is implementation-dependent.

After a write( ) to a regular file has successfully returned:

- Any successful read(2) from each byte position in the file that was modified by that write will return the data specified by the write( ) for that position until such byte positions are again modified.
- Any subsequent successful write( ) to the same byte position in the file will overwrite that file data.

Write requests to a pipe or FIFO are handled the same as a regular file with the following exceptions:

- There is no file offset associated with a pipe, hence each write request appends to the end of the pipe.
- Write requests of {PIPE_BUF} bytes or less are guaranteed not to be interleaved with data from other processes doing writes on the same pipe. Writes of greater than {PIPE_BUF} bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the O_NONBLOCK or O_NDELAY flags are set.
- If O_NONBLOCK and O_NDELAY are clear, a write request may cause the process to block, but on normal completion it returns *nbyte* .
- If O_NONBLOCK and O_NDELAY are set, write( ) does not block the process. If a write( ) request for PIPE_BUF or fewer bytes succeeds completely write( ) returns *nbyte* . Otherwise, if O_NONBLOCK is set, it returns –1 and sets errno to EAGAIN or if O_NDELAY is set, it returns 0 .

A `write( )` request for greater than `{PIPE_BUF}` bytes transfers what it can and returns the number of bytes written or it transfers no data and, if `O_NONBLOCK` is set, returns `–1` with `errno` set to `EAGAIN` or if `O_NDELAY` is set, it returns `0` . Finally, if a request is greater than `PIPE_BUF` bytes and all data previously written to the pipe has been read, `write( )` transfers at least `PIPE_BUF` bytes.

When attempting to write to a file descriptor (other than a pipe, a FIFO , a socket, or a STREAM) that supports nonblocking writes and cannot accept the data immediately:

- If `O_NONBLOCK` and `O_NDELAY` are clear, `write( )` blocks until the data can be accepted.
- If `O_NONBLOCK` or `O_NDELAY` is set, `write( )` does not block the process. If some data can be written without blocking the process, `write( )` writes what it can and returns the number of bytes written. Otherwise, if `O_NONBLOCK` is set, it returns `–1` and sets `errno` to `EAGAIN` or if `O_NDELAY` is set, it returns `0` .

Upon successful completion, where *nbyte* is greater than 0, `write( )` will mark for update the `st_ctime` and `st_mtime` fields of the file, and if the file is a regular file, the `S_ISUID` and `S_ISGID` bits of the file mode may be cleared.

For STREAMS files (see `intro`(3) and `streamio`(7I)), the operation of `write( )` is determined by the values of the minimum and maximum *nbyte* range (“packet size”) accepted by the STREAM. These values are contained in the topmost STREAM module, and 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 non-zero, `write( )` fails and sets `errno` to `ERANGE` . Writing a zero-length buffer (*nbyte* is zero) to a STREAMS device sends a zero length message with zero returned. However, writing a zero-length buffer to a pipe or FIFO sends no message and zero is returned. The user program may issue the `I_SWROPT` `ioctl`(2) to enable zero-length messages to be sent across the pipe or FIFO (see `streamio`(7I))..

When writing to a STREAM, data messages are created with a priority band of zero. When writing to a socket or to a STREAM that is not a pipe or a FIFO :

- If `O_NDELAY` and `O_NONBLOCK` are not set, and the STREAM cannot accept data (the STREAM write queue is full due to internal flow control conditions), `write( )` blocks until data can be accepted.
- If `O_NDELAY` or `O_NONBLOCK` is set and the STREAM cannot accept data, `write( )` returns `–1` and sets `errno` to `EAGAIN` .

- If O_NDELAY or O_NONBLOCK is set and part of the buffer has already been written when a condition occurs in which the STREAM cannot accept additional data, write( ) terminates and returns the number of bytes written.

The write( ) and writev( ) functions will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of errno does not reflect the result of write( ) or writev( ) but reflects the prior error.

pwrite( ) The pwrite( ) function performs the same action as write( ) , except that it writes into a given position without changing the file pointer. The first three arguments to pwrite( ) are the same as write( ) with the addition of a fourth argument *offset* for the desired position inside the file.

writev( ) The writev( ) function performs the same action as write( ) , but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: *iov* [0], *iov* [1], …, *iov* [*iovcnt* −1]. The *iovcnt* buffer is valid if greater than 0 and less than or equal to IOV_MAX . See intro(2) for a definition of IOV_MAX .

The iovec structure contains the following members:

```
caddr_t  iov_base;
int      iov_len;
```

Each iovec entry specifies the base address and length of an area in memory from which data should be written. The writev( ) function always writes all data from an area before proceeding to the next.

If *fildes* refers to a regular file and all of the iov_len members in the array pointed to by *iov* are 0, writev( ) will return 0 and have no other effect. For other file types, the behavior is unspecified.

If the sum of the iov_len values is greater than SSIZE_MAX , the operation fails and no data is transferred.

writel( ), pwritel( ), and writevl( ) 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* has this restriction:

- If the descriptor refers to a file or a FIFO , then the sensitivity label portion of *label_p* is ignored.

In all other respects, the writel( ) , pwritel( ) , and writevl( ) interfaces are analogous to the write( ) , pwrite( ) , and writev( ) interfaces.

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.

RETURN VALUES

Upon successful completion, write( ) returns the number of bytes actually written to the file associated with *fildes* . This number is never greater than *nbyte* . Otherwise, –1 is returned, the file-pointer remains unchanged, and errno is set to indicate the error.

Upon successful completion, writev( ) returns the number of bytes actually written. Otherwise, it returns –1 , the file-pointer remains unchanged, and errno is set to indicate an error.

ERRORS

The write( ) , pwrite( ) , writev( ) writel( ) , pwritel( ) , and writevl( ) functions will fail if:

EAGAIN — Mandatory file / record locking is set, O_NDELAY or O_NONBLOCK is set, and there is a blocking record lock; total amount of system memory available when reading using raw I / O is temporarily insufficient; an attempt is made to write to a STREAM that can not accept data with the O_NDELAY or O_NONBLOCK flag set; or a write to a pipe or FIFO of PIPE_BUF bytes or less is requested and less than *nbytes* of free space is available.

EBADF — The *fildes* argument is not a valid file descriptor open for writing.

EDEADLK — The write was going to go to sleep and cause a deadlock situation to occur.

EDQUOT — The user's quota of disk blocks on the file system containing the file has been exhausted.

EFAULT — The *buf* argument points to an illegal address.

EFBIG — An attempt is made to write a file that exceeds the process's file size limit or the maximum file size (see getrlimit(2) and ulimit(2)).

The file is a regular file, *nbyte* is greater than 0, and the starting position is greater than or equal to the offset

maximum established in the file description associated with *fildes* .

EINTR — A signal was caught during the write operation and no data was transferred.

EIO — The process is in the background and is attempting to write to its controlling terminal whose TOSTOP flag is set, or the process is neither ignoring nor blocking SIGTTOU signals and the process group of the process is orphaned.

ENOLCK — Enforced record locking was enabled and LOCK_MAX regions are already locked in the system, or the system record lock table was full and the write could not go to sleep until the blocking record lock was removed.

ENOLINK — The *fildes* argument is on a remote machine and the link to that machine is no longer active.

ENOSPC — During a write to an ordinary file, there is no free space left on the device.

ENOSR — An attempt is made to write to a STREAMS with insufficient STREAMS memory resources available in the system.

ENXIO — A hangup occurred on the STREAM being written to.

EPIPE — An attempt is made to write to a pipe or a FIFO that is not open for reading by any process, or that has only one end open (or to a file descriptor created by socket(3SOCKET) , using type SOCK_STREAM that is no longer connected to a peer endpoint). A SIGPIPE signal will also be sent to the process. The process dies unless special provisions were taken to catch or ignore the signal.

ERANGE — The transfer request size was outside the range supported by the STREAMS file associated with *fildes* .

The pwrite( ) function fails and the file pointer remains unchanged if:

ESPIPE — The *fildes* argument is associated with a pipe or FIFO .

The writev( ) function will fail if:

EINVAL — The sum of the iov_len values in the *iov* array would overflow an ssize_t .

The write( ) and writev( ) functions may fail if:

EINVAL — The STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device.

ENXIO A hangup occurred on the STREAM being written to.

A write to a STREAMS file may fail if an error message has been received at the STREAM head. In this case, errno is set to the value included in the error message.

The writev( ) function may fail if:

EINVAL The *iovcnt* argument was less than or equal to 0 or greater than IOV_MAX ; one of the iov_len values in the *iov* array was negative; or the sum of the iov_len values in the *iov* array overflowed an int .

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.

USAGE

The pwrite( ) function has a transitional interface for 64-bit file offsets. See lf64(5) .

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	write( ) is Async-Signal-Safe

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.

SEE ALSO

Trusted Solaris 8 Reference Manual

chmod(2) , creat(2) , fcntl(2) , getrlimit(2) , lseek(2) , open(2) , ulimit(2) , intro(3) , send(3SOCKET)

SunOS 5.8 Reference Manual

dup(2) , ioctl(2) , pipe(2) , socket(3SOCKET) , attributes(5) , lf64(5) , streamio(7I)

NAME

write, pwrite, writev, writel, pwritel, writevl – Write on a file

SYNOPSIS

```
#include <unistd.h>
ssize_t write(int fildes, const void * buf, size_t nbyte);

ssize_t pwrite(int fildes, const void * buf, size_t nbyte, off_t offset);

#include <sys/uio.h>
ssize_t writev(int fildes, const struct iovec * iov, int iovcnt);

#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);
```

DESCRIPTION

The `write()` function attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor, *fildes* .

If *nbyte* is `0` , `write()` will return `0` and have no other results if the file is a regular file; otherwise, the results are unspecified.

On a regular file or other file capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file offset associated with *fildes* . Before successful return from `write()` , the file offset is incremented by the number of bytes actually written. On a regular file, if this incremented file offset is greater than the length of the file, the length of the file will be set to this file offset.

If the `O_SYNC` flag of the file status flags is set and *fildes* refers to a regular file, a successful `write()` does not return until the data is delivered to the underlying hardware.

If *fildes* refers to a socket, `write()` is equivalent to `send`(3SOCKET) with no flags set.

On a file not capable of seeking, writing always takes place starting at the current position. The value of a file offset associated with such a device is undefined.

If the `O_APPEND` flag of the file status flags is set, the file offset will be set to the end of the file prior to each write and no intervening file modification operation will occur between changing the file offset and the write operation.

For regular files, no data transfer will occur past the offset maximum established in the open file description with *fildes* .

A `write()` to a regular file is blocked if mandatory file / record locking is set (see `chmod`(2)), and there is a record lock owned by another process on the segment of the file to be written:

- If O_NDELAY or O_NONBLOCK is set, write( ) returns –1 and sets errno to EAGAIN .
- If O_NDELAY and O_NONBLOCK are clear, write( ) sleeps until all blocking locks are removed or the write( ) is terminated by a signal.

If a write( ) requests that more bytes be written than there is room for—for example, if the write would exceed the process file size limit (see getrlimit(2) and ulimit(2)), the system file size limit, or the free space on the device—only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write( ) of 512-bytes returns 20 . The next write( ) of a non-zero number of bytes gives a failure return (except as noted for pipes and FIFO below).

If write( ) is interrupted by a signal before it writes any data, it will return –1 with errno set to EINTR .

If write( ) is interrupted by a signal after it successfully writes some data, it will return the number of bytes written.

If the value of *nbyte* is greater than SSIZE_MAX , the result is implementation-dependent.

After a write( ) to a regular file has successfully returned:

- Any successful read(2) from each byte position in the file that was modified by that write will return the data specified by the write( ) for that position until such byte positions are again modified.
- Any subsequent successful write( ) to the same byte position in the file will overwrite that file data.

Write requests to a pipe or FIFO are handled the same as a regular file with the following exceptions:

- There is no file offset associated with a pipe, hence each write request appends to the end of the pipe.
- Write requests of {PIPE_BUF} bytes or less are guaranteed not to be interleaved with data from other processes doing writes on the same pipe. Writes of greater than {PIPE_BUF} bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the O_NONBLOCK or O_NDELAY flags are set.
- If O_NONBLOCK and O_NDELAY are clear, a write request may cause the process to block, but on normal completion it returns *nbyte* .
- If O_NONBLOCK and O_NDELAY are set, write( ) does not block the process. If a write( ) request for PIPE_BUF or fewer bytes succeeds completely write( ) returns *nbyte* . Otherwise, if O_NONBLOCK is set, it returns –1 and sets errno to EAGAIN or if O_NDELAY is set, it returns 0 .

A `write( )` request for greater than `{PIPE_BUF}` bytes transfers what it can and returns the number of bytes written or it transfers no data and, if `O_NONBLOCK` is set, returns `–1` with `errno` set to `EAGAIN` or if `O_NDELAY` is set, it returns `0` . Finally, if a request is greater than `PIPE_BUF` bytes and all data previously written to the pipe has been read, `write( )` transfers at least `PIPE_BUF` bytes.

When attempting to write to a file descriptor (other than a pipe, a FIFO , a socket, or a STREAM) that supports nonblocking writes and cannot accept the data immediately:

- If `O_NONBLOCK` and `O_NDELAY` are clear, `write( )` blocks until the data can be accepted.
- If `O_NONBLOCK` or `O_NDELAY` is set, `write( )` does not block the process. If some data can be written without blocking the process, `write( )` writes what it can and returns the number of bytes written. Otherwise, if `O_NONBLOCK` is set, it returns `–1` and sets `errno` to `EAGAIN` or if `O_NDELAY` is set, it returns `0` .

Upon successful completion, where *nbyte* is greater than 0, `write( )` will mark for update the `st_ctime` and `st_mtime` fields of the file, and if the file is a regular file, the `S_ISUID` and `S_ISGID` bits of the file mode may be cleared.

For STREAMS files (see `intro`(3) and `streamio`(7I)), the operation of `write( )` is determined by the values of the minimum and maximum *nbyte* range (“packet size”) accepted by the STREAM. These values are contained in the topmost STREAM module, and 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 non-zero, `write( )` fails and sets `errno` to `ERANGE` . Writing a zero-length buffer (*nbyte* is zero) to a STREAMS device sends a zero length message with zero returned. However, writing a zero-length buffer to a pipe or FIFO sends no message and zero is returned. The user program may issue the `I_SWROPT` `ioctl`(2) to enable zero-length messages to be sent across the pipe or FIFO (see `streamio`(7I))..

When writing to a STREAM, data messages are created with a priority band of zero. When writing to a socket or to a STREAM that is not a pipe or a FIFO :

- If `O_NDELAY` and `O_NONBLOCK` are not set, and the STREAM cannot accept data (the STREAM write queue is full due to internal flow control conditions), `write( )` blocks until data can be accepted.
- If `O_NDELAY` or `O_NONBLOCK` is set and the STREAM cannot accept data, `write( )` returns `–1` and sets `errno` to `EAGAIN` .

- If O_NDELAY or O_NONBLOCK is set and part of the buffer has already been written when a condition occurs in which the STREAM cannot accept additional data, write( ) terminates and returns the number of bytes written.

The write( ) and writev( ) functions will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of errno does not reflect the result of write( ) or writev( ) but reflects the prior error.

pwrite( ) The pwrite( ) function performs the same action as write( ) , except that it writes into a given position without changing the file pointer. The first three arguments to pwrite( ) are the same as write( ) with the addition of a fourth argument *offset* for the desired position inside the file.

writev( ) The writev( ) function performs the same action as write( ) , but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: *iov* [0], *iov* [1], …, *iov* [*iovcnt* −1]. The *iovcnt* buffer is valid if greater than 0 and less than or equal to IOV_MAX . See intro(2) for a definition of IOV_MAX .

The iovec structure contains the following members:

```
caddr_t  iov_base;
int      iov_len;
```

Each iovec entry specifies the base address and length of an area in memory from which data should be written. The writev( ) function always writes all data from an area before proceeding to the next.

If *fildes* refers to a regular file and all of the iov_len members in the array pointed to by *iov* are 0, writev( ) will return 0 and have no other effect. For other file types, the behavior is unspecified.

If the sum of the iov_len values is greater than SSIZE_MAX , the operation fails and no data is transferred.

writel( ), pwritel( ), and writevl( ) 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* has this restriction:

- If the descriptor refers to a file or a FIFO , then the sensitivity label portion of *label_p* is ignored.

In all other respects, the writel( ) , pwritel( ) , and writevl( ) interfaces are analogous to the write( ) , pwrite( ) , and writev( ) interfaces.

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.

RETURN VALUES

Upon successful completion, write( ) returns the number of bytes actually written to the file associated with *fildes* . This number is never greater than *nbyte* . Otherwise, –1 is returned, the file-pointer remains unchanged, and errno is set to indicate the error.

Upon successful completion, writev( ) returns the number of bytes actually written. Otherwise, it returns –1 , the file-pointer remains unchanged, and errno is set to indicate an error.

ERRORS

The write( ) , pwrite( ) , writev( ) writel( ) , pwritel( ) , and writevl( ) functions will fail if:

EAGAIN — Mandatory file / record locking is set, O_NDELAY or O_NONBLOCK is set, and there is a blocking record lock; total amount of system memory available when reading using raw I / O is temporarily insufficient; an attempt is made to write to a STREAM that can not accept data with the O_NDELAY or O_NONBLOCK flag set; or a write to a pipe or FIFO of PIPE_BUF bytes or less is requested and less than *nbytes* of free space is available.

EBADF — The *fildes* argument is not a valid file descriptor open for writing.

EDEADLK — The write was going to go to sleep and cause a deadlock situation to occur.

EDQUOT — The user's quota of disk blocks on the file system containing the file has been exhausted.

EFAULT — The *buf* argument points to an illegal address.

EFBIG — An attempt is made to write a file that exceeds the process's file size limit or the maximum file size (see getrlimit(2) and ulimit(2)).

The file is a regular file, *nbyte* is greater than 0, and the starting position is greater than or equal to the offset

maximum established in the file description associated with *fildes* .

EINTR — A signal was caught during the write operation and no data was transferred.

EIO — The process is in the background and is attempting to write to its controlling terminal whose TOSTOP flag is set, or the process is neither ignoring nor blocking SIGTTOU signals and the process group of the process is orphaned.

ENOLCK — Enforced record locking was enabled and LOCK_MAX regions are already locked in the system, or the system record lock table was full and the write could not go to sleep until the blocking record lock was removed.

ENOLINK — The *fildes* argument is on a remote machine and the link to that machine is no longer active.

ENOSPC — During a write to an ordinary file, there is no free space left on the device.

ENOSR — An attempt is made to write to a STREAMS with insufficient STREAMS memory resources available in the system.

ENXIO — A hangup occurred on the STREAM being written to.

EPIPE — An attempt is made to write to a pipe or a FIFO that is not open for reading by any process, or that has only one end open (or to a file descriptor created by socket(3SOCKET) , using type SOCK_STREAM that is no longer connected to a peer endpoint). A SIGPIPE signal will also be sent to the process. The process dies unless special provisions were taken to catch or ignore the signal.

ERANGE — The transfer request size was outside the range supported by the STREAMS file associated with *fildes* .

The pwrite( ) function fails and the file pointer remains unchanged if:

ESPIPE — The *fildes* argument is associated with a pipe or FIFO .

The writev( ) function will fail if:

EINVAL — The sum of the iov_len values in the *iov* array would overflow an ssize_t .

The write( ) and writev( ) functions may fail if:

EINVAL — The STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device.

ENXIO A hangup occurred on the STREAM being written to.

A write to a STREAMS file may fail if an error message has been received at the STREAM head. In this case, errno is set to the value included in the error message.

The writev( ) function may fail if:

EINVAL The *iovcnt* argument was less than or equal to 0 or greater than IOV_MAX ; one of the iov_len values in the *iov* array was negative; or the sum of the iov_len values in the *iov* array overflowed an int .

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.

USAGE

The pwrite( ) function has a transitional interface for 64-bit file offsets. See lf64(5) .

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	write( ) is Async-Signal-Safe

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.

SEE ALSO

Trusted Solaris 8 Reference Manual

chmod(2) , creat(2) , fcntl(2) , getrlimit(2) , lseek(2) , open(2) , ulimit(2) , intro(3) , send(3SOCKET)

SunOS 5.8 Reference Manual

dup(2) , ioctl(2) , pipe(2) , socket(3SOCKET) , attributes(5) , lf64(5) , streamio(7I)

NAME

write, pwrite, writev, writel, pwritel, writevl – Write on a file

SYNOPSIS

```
#include <unistd.h>
ssize_t write(int fildes, const void * buf, size_t nbyte);

ssize_t pwrite(int fildes, const void * buf, size_t nbyte, off_t offset);

#include <sys/uio.h>
ssize_t writev(int fildes, const struct iovec * iov, int iovcnt);

#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);
```

DESCRIPTION

The `write()` function attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor, *fildes* .

If *nbyte* is `0` , `write()` will return `0` and have no other results if the file is a regular file; otherwise, the results are unspecified.

On a regular file or other file capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file offset associated with *fildes* . Before successful return from `write()` , the file offset is incremented by the number of bytes actually written. On a regular file, if this incremented file offset is greater than the length of the file, the length of the file will be set to this file offset.

If the `O_SYNC` flag of the file status flags is set and *fildes* refers to a regular file, a successful `write()` does not return until the data is delivered to the underlying hardware.

If *fildes* refers to a socket, `write()` is equivalent to `send`(3SOCKET) with no flags set.

On a file not capable of seeking, writing always takes place starting at the current position. The value of a file offset associated with such a device is undefined.

If the `O_APPEND` flag of the file status flags is set, the file offset will be set to the end of the file prior to each write and no intervening file modification operation will occur between changing the file offset and the write operation.

For regular files, no data transfer will occur past the offset maximum established in the open file description with *fildes* .

A `write()` to a regular file is blocked if mandatory file / record locking is set (see `chmod`(2)), and there is a record lock owned by another process on the segment of the file to be written:

- If O_NDELAY or O_NONBLOCK is set, write( ) returns –1 and sets errno to EAGAIN .
- If O_NDELAY and O_NONBLOCK are clear, write( ) sleeps until all blocking locks are removed or the write( ) is terminated by a signal.

If a write( ) requests that more bytes be written than there is room for—for example, if the write would exceed the process file size limit (see getrlimit(2) and ulimit(2)), the system file size limit, or the free space on the device—only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write( ) of 512-bytes returns 20 . The next write( ) of a non-zero number of bytes gives a failure return (except as noted for pipes and FIFO below).

If write( ) is interrupted by a signal before it writes any data, it will return –1 with errno set to EINTR .

If write( ) is interrupted by a signal after it successfully writes some data, it will return the number of bytes written.

If the value of *nbyte* is greater than SSIZE_MAX , the result is implementation-dependent.

After a write( ) to a regular file has successfully returned:

- Any successful read(2) from each byte position in the file that was modified by that write will return the data specified by the write( ) for that position until such byte positions are again modified.
- Any subsequent successful write( ) to the same byte position in the file will overwrite that file data.

Write requests to a pipe or FIFO are handled the same as a regular file with the following exceptions:

- There is no file offset associated with a pipe, hence each write request appends to the end of the pipe.
- Write requests of {PIPE_BUF} bytes or less are guaranteed not to be interleaved with data from other processes doing writes on the same pipe. Writes of greater than {PIPE_BUF} bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the O_NONBLOCK or O_NDELAY flags are set.
- If O_NONBLOCK and O_NDELAY are clear, a write request may cause the process to block, but on normal completion it returns *nbyte* .
- If O_NONBLOCK and O_NDELAY are set, write( ) does not block the process. If a write( ) request for PIPE_BUF or fewer bytes succeeds completely write( ) returns *nbyte* . Otherwise, if O_NONBLOCK is set, it returns –1 and sets errno to EAGAIN or if O_NDELAY is set, it returns 0 .

A `write( )` request for greater than `{PIPE_BUF}` bytes transfers what it can and returns the number of bytes written or it transfers no data and, if `O_NONBLOCK` is set, returns `–1` with `errno` set to `EAGAIN` or if `O_NDELAY` is set, it returns `0` . Finally, if a request is greater than `PIPE_BUF` bytes and all data previously written to the pipe has been read, `write( )` transfers at least `PIPE_BUF` bytes.

When attempting to write to a file descriptor (other than a pipe, a FIFO , a socket, or a STREAM) that supports nonblocking writes and cannot accept the data immediately:

- If `O_NONBLOCK` and `O_NDELAY` are clear, `write( )` blocks until the data can be accepted.
- If `O_NONBLOCK` or `O_NDELAY` is set, `write( )` does not block the process. If some data can be written without blocking the process, `write( )` writes what it can and returns the number of bytes written. Otherwise, if `O_NONBLOCK` is set, it returns `–1` and sets `errno` to `EAGAIN` or if `O_NDELAY` is set, it returns `0` .

Upon successful completion, where *nbyte* is greater than 0, `write( )` will mark for update the `st_ctime` and `st_mtime` fields of the file, and if the file is a regular file, the `S_ISUID` and `S_ISGID` bits of the file mode may be cleared.

For STREAMS files (see `intro`(3) and `streamio`(7I)), the operation of `write( )` is determined by the values of the minimum and maximum *nbyte* range (“packet size”) accepted by the STREAM. These values are contained in the topmost STREAM module, and 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 non-zero, `write( )` fails and sets `errno` to `ERANGE` . Writing a zero-length buffer (*nbyte* is zero) to a STREAMS device sends a zero length message with zero returned. However, writing a zero-length buffer to a pipe or FIFO sends no message and zero is returned. The user program may issue the `I_SWROPT` `ioctl`(2) to enable zero-length messages to be sent across the pipe or FIFO (see `streamio`(7I))..

When writing to a STREAM, data messages are created with a priority band of zero. When writing to a socket or to a STREAM that is not a pipe or a FIFO :

- If `O_NDELAY` and `O_NONBLOCK` are not set, and the STREAM cannot accept data (the STREAM write queue is full due to internal flow control conditions), `write( )` blocks until data can be accepted.
- If `O_NDELAY` or `O_NONBLOCK` is set and the STREAM cannot accept data, `write( )` returns `–1` and sets `errno` to `EAGAIN` .

- If O_NDELAY or O_NONBLOCK is set and part of the buffer has already been written when a condition occurs in which the STREAM cannot accept additional data, write( ) terminates and returns the number of bytes written.

The write( ) and writev( ) functions will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of errno does not reflect the result of write( ) or writev( ) but reflects the prior error.

pwrite( ) The pwrite( ) function performs the same action as write( ) , except that it writes into a given position without changing the file pointer. The first three arguments to pwrite( ) are the same as write( ) with the addition of a fourth argument *offset* for the desired position inside the file.

writev( ) The writev( ) function performs the same action as write( ) , but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: *iov* [0], *iov* [1], …, *iov* [*iovcnt* −1]. The *iovcnt* buffer is valid if greater than 0 and less than or equal to IOV_MAX . See intro(2) for a definition of IOV_MAX .

The iovec structure contains the following members:

```
caddr_t  iov_base;
int      iov_len;
```

Each iovec entry specifies the base address and length of an area in memory from which data should be written. The writev( ) function always writes all data from an area before proceeding to the next.

If *fildes* refers to a regular file and all of the iov_len members in the array pointed to by *iov* are 0, writev( ) will return 0 and have no other effect. For other file types, the behavior is unspecified.

If the sum of the iov_len values is greater than SSIZE_MAX , the operation fails and no data is transferred.

writel( ), pwritel( ), and writevl( ) 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* has this restriction:

- If the descriptor refers to a file or a FIFO , then the sensitivity label portion of *label_p* is ignored.

In all other respects, the writel( ) , pwritel( ) , and writevl( ) interfaces are analogous to the write( ) , pwrite( ) , and writev( ) interfaces.

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.

RETURN VALUES

Upon successful completion, write( ) returns the number of bytes actually written to the file associated with *fildes* . This number is never greater than *nbyte* . Otherwise, –1 is returned, the file-pointer remains unchanged, and errno is set to indicate the error.

Upon successful completion, writev( ) returns the number of bytes actually written. Otherwise, it returns –1 , the file-pointer remains unchanged, and errno is set to indicate an error.

ERRORS

The write( ) , pwrite( ) , writev( ) writel( ) , pwritel( ) , and writevl( ) functions will fail if:

EAGAIN — Mandatory file / record locking is set, O_NDELAY or O_NONBLOCK is set, and there is a blocking record lock; total amount of system memory available when reading using raw I / O is temporarily insufficient; an attempt is made to write to a STREAM that can not accept data with the O_NDELAY or O_NONBLOCK flag set; or a write to a pipe or FIFO of PIPE_BUF bytes or less is requested and less than *nbytes* of free space is available.

EBADF — The *fildes* argument is not a valid file descriptor open for writing.

EDEADLK — The write was going to go to sleep and cause a deadlock situation to occur.

EDQUOT — The user's quota of disk blocks on the file system containing the file has been exhausted.

EFAULT — The *buf* argument points to an illegal address.

EFBIG — An attempt is made to write a file that exceeds the process's file size limit or the maximum file size (see getrlimit(2) and ulimit(2)).

The file is a regular file, *nbyte* is greater than 0, and the starting position is greater than or equal to the offset

maximum established in the file description associated with *fildes* .

EINTR — A signal was caught during the write operation and no data was transferred.

EIO — The process is in the background and is attempting to write to its controlling terminal whose TOSTOP flag is set, or the process is neither ignoring nor blocking SIGTTOU signals and the process group of the process is orphaned.

ENOLCK — Enforced record locking was enabled and LOCK_MAX regions are already locked in the system, or the system record lock table was full and the write could not go to sleep until the blocking record lock was removed.

ENOLINK — The *fildes* argument is on a remote machine and the link to that machine is no longer active.

ENOSPC — During a write to an ordinary file, there is no free space left on the device.

ENOSR — An attempt is made to write to a STREAMS with insufficient STREAMS memory resources available in the system.

ENXIO — A hangup occurred on the STREAM being written to.

EPIPE — An attempt is made to write to a pipe or a FIFO that is not open for reading by any process, or that has only one end open (or to a file descriptor created by socket(3SOCKET) , using type SOCK_STREAM that is no longer connected to a peer endpoint). A SIGPIPE signal will also be sent to the process. The process dies unless special provisions were taken to catch or ignore the signal.

ERANGE — The transfer request size was outside the range supported by the STREAMS file associated with *fildes* .

The pwrite( ) function fails and the file pointer remains unchanged if:

ESPIPE — The *fildes* argument is associated with a pipe or FIFO .

The writev( ) function will fail if:

EINVAL — The sum of the iov_len values in the *iov* array would overflow an ssize_t .

The write( ) and writev( ) functions may fail if:

EINVAL — The STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device.

ENXIO A hangup occurred on the STREAM being written to.

A write to a STREAMS file may fail if an error message has been received at the STREAM head. In this case, errno is set to the value included in the error message.

The writev( ) function may fail if:

EINVAL The *iovcnt* argument was less than or equal to 0 or greater than IOV_MAX ; one of the iov_len values in the *iov* array was negative; or the sum of the iov_len values in the *iov* array overflowed an int .

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.

USAGE

The pwrite( ) function has a transitional interface for 64-bit file offsets. See lf64(5) .

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	write( ) is Async-Signal-Safe

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.

SEE ALSO

Trusted Solaris 8 Reference Manual

chmod(2) , creat(2) , fcntl(2) , getrlimit(2) , lseek(2) , open(2) , ulimit(2) , intro(3) , send(3SOCKET)

SunOS 5.8 Reference Manual

dup(2) , ioctl(2) , pipe(2) , socket(3SOCKET) , attributes(5) , lf64(5) , streamio(7I)

NAME

write, pwrite, writev, writel, pwritel, writevl – Write on a file

SYNOPSIS

```
#include <unistd.h>
ssize_t write(int fildes, const void * buf, size_t nbyte);

ssize_t pwrite(int fildes, const void * buf, size_t nbyte, off_t offset);

#include <sys/uio.h>
ssize_t writev(int fildes, const struct iovec * iov, int iovcnt);

#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);
```

DESCRIPTION

The `write()` function attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor, *fildes* .

If *nbyte* is `0` , `write()` will return `0` and have no other results if the file is a regular file; otherwise, the results are unspecified.

On a regular file or other file capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file offset associated with *fildes* . Before successful return from `write()` , the file offset is incremented by the number of bytes actually written. On a regular file, if this incremented file offset is greater than the length of the file, the length of the file will be set to this file offset.

If the `O_SYNC` flag of the file status flags is set and *fildes* refers to a regular file, a successful `write()` does not return until the data is delivered to the underlying hardware.

If *fildes* refers to a socket, `write()` is equivalent to `send`(3SOCKET) with no flags set.

On a file not capable of seeking, writing always takes place starting at the current position. The value of a file offset associated with such a device is undefined.

If the `O_APPEND` flag of the file status flags is set, the file offset will be set to the end of the file prior to each write and no intervening file modification operation will occur between changing the file offset and the write operation.

For regular files, no data transfer will occur past the offset maximum established in the open file description with *fildes* .

A `write()` to a regular file is blocked if mandatory file / record locking is set (see `chmod`(2)), and there is a record lock owned by another process on the segment of the file to be written:

- If O_NDELAY or O_NONBLOCK is set, write( ) returns –1 and sets errno to EAGAIN .
- If O_NDELAY and O_NONBLOCK are clear, write( ) sleeps until all blocking locks are removed or the write( ) is terminated by a signal.

If a write( ) requests that more bytes be written than there is room for—for example, if the write would exceed the process file size limit (see getrlimit(2) and ulimit(2)), the system file size limit, or the free space on the device—only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write( ) of 512-bytes returns 20 . The next write( ) of a non-zero number of bytes gives a failure return (except as noted for pipes and FIFO below).

If write( ) is interrupted by a signal before it writes any data, it will return –1 with errno set to EINTR .

If write( ) is interrupted by a signal after it successfully writes some data, it will return the number of bytes written.

If the value of *nbyte* is greater than SSIZE_MAX , the result is implementation-dependent.

After a write( ) to a regular file has successfully returned:

- Any successful read(2) from each byte position in the file that was modified by that write will return the data specified by the write( ) for that position until such byte positions are again modified.
- Any subsequent successful write( ) to the same byte position in the file will overwrite that file data.

Write requests to a pipe or FIFO are handled the same as a regular file with the following exceptions:

- There is no file offset associated with a pipe, hence each write request appends to the end of the pipe.
- Write requests of {PIPE_BUF} bytes or less are guaranteed not to be interleaved with data from other processes doing writes on the same pipe. Writes of greater than {PIPE_BUF} bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the O_NONBLOCK or O_NDELAY flags are set.
- If O_NONBLOCK and O_NDELAY are clear, a write request may cause the process to block, but on normal completion it returns *nbyte* .
- If O_NONBLOCK and O_NDELAY are set, write( ) does not block the process. If a write( ) request for PIPE_BUF or fewer bytes succeeds completely write( ) returns *nbyte* . Otherwise, if O_NONBLOCK is set, it returns –1 and sets errno to EAGAIN or if O_NDELAY is set, it returns 0 .

A `write( )` request for greater than `{PIPE_BUF}` bytes transfers what it can and returns the number of bytes written or it transfers no data and, if `O_NONBLOCK` is set, returns `–1` with `errno` set to `EAGAIN` or if `O_NDELAY` is set, it returns `0` . Finally, if a request is greater than `PIPE_BUF` bytes and all data previously written to the pipe has been read, `write( )` transfers at least `PIPE_BUF` bytes.

When attempting to write to a file descriptor (other than a pipe, a FIFO , a socket, or a STREAM) that supports nonblocking writes and cannot accept the data immediately:

- If `O_NONBLOCK` and `O_NDELAY` are clear, `write( )` blocks until the data can be accepted.
- If `O_NONBLOCK` or `O_NDELAY` is set, `write( )` does not block the process. If some data can be written without blocking the process, `write( )` writes what it can and returns the number of bytes written. Otherwise, if `O_NONBLOCK` is set, it returns `–1` and sets `errno` to `EAGAIN` or if `O_NDELAY` is set, it returns `0` .

Upon successful completion, where *nbyte* is greater than 0, `write( )` will mark for update the `st_ctime` and `st_mtime` fields of the file, and if the file is a regular file, the `S_ISUID` and `S_ISGID` bits of the file mode may be cleared.

For STREAMS files (see `intro`(3) and `streamio`(7I)), the operation of `write( )` is determined by the values of the minimum and maximum *nbyte* range (“packet size”) accepted by the STREAM. These values are contained in the topmost STREAM module, and 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 non-zero, `write( )` fails and sets `errno` to `ERANGE` . Writing a zero-length buffer (*nbyte* is zero) to a STREAMS device sends a zero length message with zero returned. However, writing a zero-length buffer to a pipe or FIFO sends no message and zero is returned. The user program may issue the `I_SWROPT` `ioctl`(2) to enable zero-length messages to be sent across the pipe or FIFO (see `streamio`(7I))..

When writing to a STREAM, data messages are created with a priority band of zero. When writing to a socket or to a STREAM that is not a pipe or a FIFO :

- If `O_NDELAY` and `O_NONBLOCK` are not set, and the STREAM cannot accept data (the STREAM write queue is full due to internal flow control conditions), `write( )` blocks until data can be accepted.
- If `O_NDELAY` or `O_NONBLOCK` is set and the STREAM cannot accept data, `write( )` returns `–1` and sets `errno` to `EAGAIN` .

- If O_NDELAY or O_NONBLOCK is set and part of the buffer has already been written when a condition occurs in which the STREAM cannot accept additional data, write( ) terminates and returns the number of bytes written.

The write( ) and writev( ) functions will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of errno does not reflect the result of write( ) or writev( ) but reflects the prior error.

pwrite( ) The pwrite( ) function performs the same action as write( ) , except that it writes into a given position without changing the file pointer. The first three arguments to pwrite( ) are the same as write( ) with the addition of a fourth argument *offset* for the desired position inside the file.

writev( ) The writev( ) function performs the same action as write( ) , but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: *iov* [0], *iov* [1], …, *iov* [*iovcnt* −1]. The *iovcnt* buffer is valid if greater than 0 and less than or equal to IOV_MAX . See intro(2) for a definition of IOV_MAX .

The iovec structure contains the following members:

```
caddr_t  iov_base;
int      iov_len;
```

Each iovec entry specifies the base address and length of an area in memory from which data should be written. The writev( ) function always writes all data from an area before proceeding to the next.

If *fildes* refers to a regular file and all of the iov_len members in the array pointed to by *iov* are 0, writev( ) will return 0 and have no other effect. For other file types, the behavior is unspecified.

If the sum of the iov_len values is greater than SSIZE_MAX , the operation fails and no data is transferred.

writel( ), pwritel( ), and writevl( ) 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* has this restriction:

- If the descriptor refers to a file or a FIFO , then the sensitivity label portion of *label_p* is ignored.

In all other respects, the writel( ) , pwritel( ) , and writevl( ) interfaces are analogous to the write( ) , pwrite( ) , and writev( ) interfaces.

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.

RETURN VALUES

Upon successful completion, write( ) returns the number of bytes actually written to the file associated with *fildes* . This number is never greater than *nbyte* . Otherwise, –1 is returned, the file-pointer remains unchanged, and errno is set to indicate the error.

Upon successful completion, writev( ) returns the number of bytes actually written. Otherwise, it returns –1 , the file-pointer remains unchanged, and errno is set to indicate an error.

ERRORS

The write( ) , pwrite( ) , writev( ) writel( ) , pwritel( ) , and writevl( ) functions will fail if:

EAGAIN — Mandatory file / record locking is set, O_NDELAY or O_NONBLOCK is set, and there is a blocking record lock; total amount of system memory available when reading using raw I / O is temporarily insufficient; an attempt is made to write to a STREAM that can not accept data with the O_NDELAY or O_NONBLOCK flag set; or a write to a pipe or FIFO of PIPE_BUF bytes or less is requested and less than *nbytes* of free space is available.

EBADF — The *fildes* argument is not a valid file descriptor open for writing.

EDEADLK — The write was going to go to sleep and cause a deadlock situation to occur.

EDQUOT — The user's quota of disk blocks on the file system containing the file has been exhausted.

EFAULT — The *buf* argument points to an illegal address.

EFBIG — An attempt is made to write a file that exceeds the process's file size limit or the maximum file size (see getrlimit(2) and ulimit(2)).

The file is a regular file, *nbyte* is greater than 0, and the starting position is greater than or equal to the offset

maximum established in the file description associated with *fildes* .

EINTR — A signal was caught during the write operation and no data was transferred.

EIO — The process is in the background and is attempting to write to its controlling terminal whose TOSTOP flag is set, or the process is neither ignoring nor blocking SIGTTOU signals and the process group of the process is orphaned.

ENOLCK — Enforced record locking was enabled and LOCK_MAX regions are already locked in the system, or the system record lock table was full and the write could not go to sleep until the blocking record lock was removed.

ENOLINK — The *fildes* argument is on a remote machine and the link to that machine is no longer active.

ENOSPC — During a write to an ordinary file, there is no free space left on the device.

ENOSR — An attempt is made to write to a STREAMS with insufficient STREAMS memory resources available in the system.

ENXIO — A hangup occurred on the STREAM being written to.

EPIPE — An attempt is made to write to a pipe or a FIFO that is not open for reading by any process, or that has only one end open (or to a file descriptor created by socket(3SOCKET) , using type SOCK_STREAM that is no longer connected to a peer endpoint). A SIGPIPE signal will also be sent to the process. The process dies unless special provisions were taken to catch or ignore the signal.

ERANGE — The transfer request size was outside the range supported by the STREAMS file associated with *fildes* .

The pwrite( ) function fails and the file pointer remains unchanged if:

ESPIPE — The *fildes* argument is associated with a pipe or FIFO .

The writev( ) function will fail if:

EINVAL — The sum of the iov_len values in the *iov* array would overflow an ssize_t .

The write( ) and writev( ) functions may fail if:

EINVAL — The STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device.

ENXIO A hangup occurred on the STREAM being written to.

A write to a STREAMS file may fail if an error message has been received at the STREAM head. In this case, errno is set to the value included in the error message.

The writev( ) function may fail if:

EINVAL The *iovcnt* argument was less than or equal to 0 or greater than IOV_MAX ; one of the iov_len values in the *iov* array was negative; or the sum of the iov_len values in the *iov* array overflowed an int .

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.

USAGE

The pwrite( ) function has a transitional interface for 64-bit file offsets. See lf64(5) .

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	write( ) is Async-Signal-Safe

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.

SEE ALSO

Trusted Solaris 8 Reference Manual

chmod(2) , creat(2) , fcntl(2) , getrlimit(2) , lseek(2) , open(2) , ulimit(2) , intro(3) , send(3SOCKET)

SunOS 5.8 Reference Manual

dup(2) , ioctl(2) , pipe(2) , socket(3SOCKET) , attributes(5) , lf64(5) , streamio(7I)

Index

E

F

G

H

R

S

T

U

V

W