NAME

Intro, intro – introduction to system calls and error numbers

SYNOPSIS

#include <errno.h>

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

37 **ECHRNG** Channel number out of range

38 **EL2NSYNC** Level 2 not synchronized

39 **EL3HLT** Level 3 halted

40 **EL3RST** Level 3 reset

41 **ELNRNG** Link number out of range

42 **EUNATCH** Protocol driver not attached

43 **ENOCSI** No CSI structure available

44 **EL2HLT** Level 2 halted

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

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

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

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

49 Reserved

58–59 Reserved

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

61 **ENODATA** No data available

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

82 **EREMCHG** Remote address changed

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

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

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

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

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

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

89 **ENOSYS** Operation not applicable

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

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

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

93 **ENOTEMPTY** Directory not empty

94 **EUSERS** Too many users

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

151 **ESTALE** Stale NFS file handle

DEFINITIONS

Background Process Group

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

Controlling Process

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

Controlling Terminal

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

Directory

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

Downstream

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

Driver

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

Effective User ID and Effective Group ID

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

File Access Permissions

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

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

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

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

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

Otherwise, the corresponding permissions are denied.

File Descriptor

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

File Name

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

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

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

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

Foreground Process Group

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

{IOV_MAX}

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

{LIMIT}

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

Masks

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

Message

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

Message Queue

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

Message Queue Identifier

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

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

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

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

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

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

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

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

msg_cbytes is the current number of bytes on the queue.

msg_qnum is the number of messages currently on the queue.

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

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

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

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

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

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

Message Operation Permissions

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

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

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

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

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

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

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

Otherwise, the corresponding permissions are denied.

Module A module is an entity containing processing routines for input and output data. It always exists in the middle of a stream, between the stream’s head and a driver. A module is the STREAMS counterpart to the commands in a shell pipeline except that a module contains a pair of functions which allow independent bidirectional (downstream and upstream) data flow and processing.

Multiplexor A multiplexor is a driver that allows streams associated with several user processes to be connected to a single driver, or several drivers to be connected to a single user process. STREAMS does not provide a general multiplexing driver, but does provide the facilities for constructing them and for connecting multiplexed configurations of streams.

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

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

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

A slash by itself names the root directory.

Unless specifically stated otherwise, the null path name is treated as if it named a 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 processs, there are threads with thread id’s, called thread_t and LWPID_t. These threads are not visable to the outside process.

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

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

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

Process Group Leader

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

Process Group ID

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

Process Lifetime

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

Process Group Lifetime

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

Read Queue

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

Real User ID and Real Group ID

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

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

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

Root Directory and Current Working Directory

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

Saved User ID and Saved Group ID

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

Semaphore Identifier

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Semaphore Operation Permissions

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

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

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

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

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

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

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

Otherwise, the corresponding permissions are denied.

Session

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

Session ID

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

Session Leader

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

Session Lifetime

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

Shared Memory Identifier

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

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

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

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

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

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

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

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

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

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

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

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

Shared Memory Operation Permissions

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

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

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

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

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

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

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

Otherwise, the corresponding permissions are denied.

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

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

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

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

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

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

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

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

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

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

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

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

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

yield(2)	yield execution to another lightweight process