SCO Streams

Toolkit

Release and Installation Notes

Version 1.0

The Santa Cruz Operation

This document was typeset with an IMAGEN® 8/300 Laser Printer.

SCO Document Number: XG-11-1-88-1.0A
Printed: Fri Dec 2 14:58:09 PST 1988

SCO Streams Toolkit Release and Installation Notes Version 1.0

SCO Streams
Toolkit
Version 1.0
Release and Installation Notes
November 1, 1988

1. Preface

These notes explain how to install the SCO Streams Toolkit on a 386 computer running SCO XENIX System V, release 2.3 or higher. Do not attempt this installation on a computer running an earlier version of XENIX, or on a 286 computer.

Manual pages for the following commands are included with these Release and Installation Notes:

close (S)
dup (S)
exec (S)
exit (S)
fcntl (S)
getmsg (S)
ioctl (S)
open (S)
poll (S)
putmsg (S)
read (S)
signal (S)
sigset (S)
write (S)

Four of these commands are new (**getmsg**, **poll**, **putmsg**, and **sigset**). The remainder have new features or references related to STREAMS. You should place these manual pages in section S of your existing reference manual. If the command is new, add it to the section; if it is already present, replace the existing manual pages with the new set.

2. Installing the Streams Toolkit

Follow the steps outlined below to install the SCO Streams Toolkit:

1. As *root*, activate *custom* by entering:

   custom

2. Select *Add a Supported Product*

3. You are prompted to insert distribution volume 1. Insert the streams diskette and press <Return>.

4. You see a menu. Select *Install one or more packages.*

5. You see another menu. Enter **toolkit** and press <Return>.

6. You are prompted to insert the SCO Streams Toolkit volume 1. Press <Return>. Files are extracted.

7. You see copyright information and are prompted to enter your serial number. Enter your SCO Streams Toolkit serial number and press <Return>.

8. You are prompted for your activation key. Enter your SCO Streams Toolkit activation key and press <Return>.

9. You are returned to a *custom* menu. Press <q> to return to the XENIX prompt.

The installation of the Streams Toolkit is complete.

3. Removing the Streams Toolkit

Follow the steps outlined below to remove the SCO Streams Runtime System from your computer:

1. As *root*, activate *custom* by entering the following command:

   custom

2. Select *SCO Streams Toolkit.*

3. Select *Remove one or more packages.*

4. Enter **toolkit** and press <Return>.

5. You are returned to a *custom* menu. Press <q> to return to the XENIX prompt.

4. Distribution

The Streams Toolkit diskette contains the following files:

./tmp/perms/streamstk
./usr/include/poll.h
./usr/include/stropts.h
./usr/include/tiuser.h
./usr/include/sys/poll.h
./usr/include/sys/stream.h
./usr/include/sys/strlog.h
./usr/include/sys/stropts.h
./usr/include/sys/strstat.h
./usr/include/sys/tihdr.h
./usr/include/sys/timod.h
./usr/include/sys/tiuser.h
./lib/386/Slibnsl.a
./tmp/init.streamst

Name

close - close a file descriptor

Syntax

int close (fildes)
int fildes;

Description

The *fildes* argument is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. The *close* system call closes the file descriptor indicated by *fildes*. All outstanding record locks owned by the process (on the file indicated by *fildes*) are removed.

If a STREAMS [see *intro*(S)] file is closed, and the calling process had previously registered to receive a SIGPOLL signal [see *signal*(S) and *sigset*(S)] for events associated with that file [see I_SETSIG in *streamio*(STR)], the calling process will be unregistered for events associated with the file. The last *close* for a *stream* causes the *stream* associated with *fildes* to be dismantled. If O_NDELAY is not set and there have been no signals posted for the *stream*, *close* waits up to 15 seconds, for each module and driver, for any output to drain before dismantling the *stream*. If the O_NDELAY flag is set or if there are any pending signals, *close* does not wait for output to drain and dismantles the *stream* immediately.

The named file is closed unless one or more of the following is true:

[EBADF]	The *fildes* argument is not a valid open file descriptor.
[EINTR]	A signal was caught during the *close* system call.
[ENOLINK]	*fildes* is on a remote machine and the link to that machine is no longer active.

See Also

creat(S), dup(S), exec(S), fcntl(S), intro(S), open(S), pipe(S), signal(S), sigset(S).
streamio(STR) in Appendix F of the *STREAMS Programmer's Guide*.

Diagnostics

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

Name

dup - duplicate an open file descriptor

Syntax

int dup (fildes)
int fildes;

Description

The *fildes* argument is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. The *dup* system call returns a new file descriptor having the following in common with the original:

Same open file (or pipe)

Same file pointer (that is, both file descriptors share one file pointer)

Same access mode (read, write, or read/write)

The new file descriptor is set to remain open across *exec* system calls [see *fcntl* (S)].

The file descriptor returned is the lowest one available.

The *dup* system call will fail if one or more of the following is true:

[EBADF]	The *fildes* argument is not a valid open file descriptor.
[EINTR]	A signal was caught during the *dup* system call.
[EMFILE]	NOFILES file descriptors are currently open.
[ENOLINK]	*Fildes* is on a remote machine and the link to that machine is no longer active.

See Also

close(S), creat(S), exec(S), fcntl(S), open(S), pipe(S), lockf(S).

Diagnostics

Upon successful completion a non-negative integer, namely the file descriptor, is returned. Otherwise, a value of -1 is returned, and *errno* is set to indicate the error.

Name

exec: execl, execv, execle, execve, execlp, execvp - execute a file

Syntax

```
int execl (path, arg0, arg1, ..., argn, (char *)0)
char *path, *arg0, *arg1, ..., *argn;

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

int execle (path, arg0, arg1, ..., argn, (char *)0, envp)
char *path, *arg0, *arg1, ..., *argn, *envp[ ];

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

int execlp (file, arg0, arg1, ..., argn, (char *)0)
char *file, *arg0, *arg1, ..., *argn;

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

Descripton

The *exec* system call in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary, executable file called the *new process file*. This file consists of a header [see *a.out*(F)], a text segment, and a data segment. The data segment contains an initialized portion and an uninitialized portion (bss). There can be no return from a successful *exec* because the calling process is overlaid by the new process.

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

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

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

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

The *file* argument points to the new process file. The path prefix for this file is obtained by a search of the directories passed as the *environment* line "PATH =" [see *environ*(M)]. The environment is supplied by the shell [see *sh*(C)].

arg0, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least *arg0* must be present and point to a string that is the same as *path* (or its last component).

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

envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process. *envp* is terminated by a null pointer. For *execl* and *execv*, the C run-time start-off routine places a pointer to the environment of the calling process in the global cell:

extern char **environ;

and it is used to pass the environment of the calling process to the new process.

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

Signals set to terminate the calling process will be set to terminate the new process. Signals set to be ignored by the calling process will be set to be ignored by the new process. Signals set to be caught by the calling process will be set to terminate the new process; see *signal*(S).

For signals set by *sigset*(S), *exec* will ensure that the new process has the same system signal action for each signal type whose action is SIG_DFL, SIG_IGN, or SIG_HOLD as the calling process. However, if the action is to catch the signal, then the action will be reset to SIG_DFL, and any pending signal for this type will be held.

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

The shared memory segments attached to the calling process will not be attached to the new process [see *shmop*(S)].

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

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

nice value [see *nice*(S)]
process ID
parent process ID
process group ID
semadj values [see *semop*(S)]
tty group ID [see *exit*(S) and *signal*(S)]
trace flag [see *ptrace*(S) request 0]
time left until an alarm clock signal [see *alarm*(S)]
current working directory
root directory
file mode creation mask [see *umask*(S)]
file size limit [see *ulimit*(S)]
utime, *stime*, *cutime*, and *cstime* [see *times*(S)]
file-locks [see *fcntl*(S) and *lockf*(S)]

The *exec* system call will fail and return to the calling process if one or more of the following is true:

[ENOENT] One or more components of the new process path name of the file do not exist.

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

[EACCES] Search permission is denied for a directory listed in the new process file's path prefix.

[EACCES] The new process file is not an ordinary file.

[EACCES] The new process file mode denies execution permission.

[ENOEXEC] The exec is not an *execlp* or *execvp*, and the new process file has the appropriate access permission but an invalid magic number in its header.

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

[ENOMEM] The new process requires more memory than is allowed by the system-imposed maximum MAXMEM.

[E2BIG]	The number of bytes in the new process's argument list is greater than the system-imposed limit of 5120 bytes.
[EFAULT]	Required hardware is not present.
[EFAULT]	*path*, *argv*, or *envp* point to an illegal address.
[EAGAIN]	Not enough memory.
[ELIBACC]	Required shared library does not have execute permission.
[ELIBEXEC]	Trying to *exec*(S) a shared library directly.
[EINTR]	A signal was caught during the *exec* system call.
[ENOLINK]	*path* points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.

See Also

alarm(S), exit(S), fcntl(S), fork(S), nice(S), ptrace(S), semop(S), signal(S), sigset(S), times(S), ulimit(S), umask(S), lockf(S).
a.out(F), environ(M), sh(C) in the *XENIX Reference*.

Diagnostics

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

Name

exit, _exit - terminate process

Syntax

void exit (status)
int status;
void _exit (status)
int status;

Description

The *exit* system call terminates the calling process with the following consequences:

All of the file descriptors open in the calling process are closed.

If the parent process of the calling process is executing a *wait*, it is notified of the calling process's termination and the low order eight bits (that is, bits 0377) of *status* are made available to it [see *wait*(S)].

If the parent process of the calling process is not executing a *wait*, the calling process is transformed into a zombie process. A *zombie process* is a process that only occupies a slot in the process table. It has no other space allocated either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information (see **<sys/proc.h>**) to be used by *times*.

The parent process ID of all of the calling processes' existing child processes and zombie processes is set to 1. This means the initialization process [see *intro*(S)] inherits each of these processes.

Each attached shared memory segment is detached and the value of **shm_nattach** in the data structure associated with its shared memory identifier is decremented by 1.

For each semaphore for which the calling process has set a semadj value [see *semop*(S)], that semadj value is added to the semval of the specified semaphore.

If the process has a process, text, or data lock, an *unlock* is performed [see *plock*(S)].

An accounting record is written on the accounting file if the system's accounting routine is enabled [see *acct* (S)].

If the process ID, tty group ID, and process group ID of the calling process are equal, the **SIGHUP** signal is sent to each process that has a process group ID equal to that of the calling process.

A death of child signal is sent to the parent.

The C function *exit* may cause cleanup actions before the process exits. The function *_exit* circumvents all cleanup.

See Also

acct(S), intro(S), plock(S), semop(S), signal(S), sigset(S), wait(S).

Diagnostics

None. There can be no return from an *exit* system call.

Name

fcntl - file control

Syntax

#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd;

Description

The *fcntl* system call provides for control over open files. The *fildes* argument is an open file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. The data type and value of *arg* are specific to the type of command specified by *cmd*. The symbolic names for commands and file status flags are defined by the **<fcntl.h>** header file.

The commands available are:

F_DUPFD Return a new file descriptor as follows:

Lowest numbered available file descriptor greater than or equal to *arg*.

Same open file (or pipe) as the original file.

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

Same access mode (read, write, or read/write).

Same file status flags (that is, both file descriptors share the same file status flags).

The close-on-exec flag associated with the new file descriptor is set to remain open across *exec* (S) system calls.

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

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

F_GETFL Get *file* status flags [see *open*(S)].

F_SETFL Set *file* status flags to *arg*. Only certain flags can be set (See the include file **<fcntl.h>**).

The following commands are used for file-locking and record-locking. Locks may be placed on an entire file or segments of a file.

F_GETLK
Get the first lock that blocks the lock description given by the variable of type *struct flock* pointed to by *arg*. The information retrieved overwrites the information passed to *fcntl* in the *flock* structure. If no lock is found that would prevent this lock from being created, then the structure is passed back unchanged except for the lock type which will be set to F_UNLCK.

F_SETLK
Set or clear a file segment lock according to the variable of type *struct flock* pointed to by *arg* (see the include file **<fcntl.h>**). The *cmd* F_SETLK is used to establish read (F_RDLCK) and write (F_WRLCK) locks, as well as remove either type of lock (F_UNLCK). If a read or write lock cannot be set, *fcntl* will return immediately with an error value of -1.

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

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

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

The structure *flock* defined in the **<fcntl.h>** header file describes a lock. It describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), size (*l_len*), and process-ID (*l_pid*):

```
short l_type;     /* F_RDLCK, F_WRLCK, F_UNLCK */
short l_whence;   /* flag for starting offset */
long l_start;     /* relative offset in bytes */
long l_len;       /* if 0 then until EOF */
short l_pid;      /* returned with F_GETLK */
```

The value of *l_whence* is 0, 1, or 2 to indicate that the relative offset, *l_start* bytes, will be measured from the start of the file, current position, or end of file, respectively. The value of *l_len* is the number of

consecutive bytes to be locked. The process id is used only with the F_GETLK *cmd* to return the values for a blocking lock. Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end of file by setting *l_len* to zero (0). If such a lock also has *l_whence* and *l_start* set to zero (0), the whole file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments for either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. 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 in a *fork*(S) system call.

When mandatory file and record locking is active on a file [see *chmod*(S)], *read* and *write* system calls issued on the file will be affected by the record locks in effect.

The *fcntl* system call will fail if one or more of the following is true:

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

[EINVAL] The *cmd* argument is F_DUPFD. The *arg* argument is either negative, or greater than or equal to the configured value for the maximum number of open file descriptors allowed each user.

[EINVAL] The *cmd* argument is F_GETLK, F_SETLK, or SETLKW and *arg* or the data it points to is not valid.

[EACCES] The *cmd* argument is F_SETLK, the type of lock (*l_type*) is a read (F_RDLCK) lock, and the segment of a file to be locked is already write locked by another process or the type is a write (F_WRLCK) lock and the segment of a file to be locked is already read or write locked by another process.

[ENOLCK] The *cmd* argument is F_SETLK or F_SETLKW, the type of lock is a read or write lock, and there are no more record locks available (too many file segments locked) because the system maximum has been exceeded.

[EMFILE] The *cmd* argument is F_DUPFD and file-descriptors are currently open in the calling-process.

[EBADF] The *cmd* argument is F_SETLK or F_SETLKW, the type of lock (*l_type*) is a read-lock (F_RDLCK), and *fildes* is not a valid file-descriptor open for reading.

[EBADF] The *cmd* argument is F_SETLK or F_SETLKW, the type of lock (*l_type*) is a write-lock (F_WRLCK), and *fildes* is not a valid file-descriptor open for writing.

[EDEADLK] The *cmd* argument is F_SETLKW, 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.

[EFAULT] The *cmd* argument is F_SETLK, *arg* points outside the program address space.

[EINTR] A signal was caught during the *fcntl* system call.

[ENOLINK] *fildes* is on a remote machine and the link to that machine is no longer active.

See Also

close(S), creat(S), dup(S), exec(S), fork(S), open(S), pipe(S).

Diagnostics

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

F_DUPFD	A new file descriptor.
F_GETFD	Value of flag (only the low-order bit is defined).
F_SETFD	Value other than -1.
F_GETFL	Value of file flags.
F_SETFL	Value other than -1.
F_GETLK	Value other than -1.
F_SETLK	Value other than -1.
F_SETLKW	Value other than -1.

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

Warning

Because in the future the variable *errno* will be set to EAGAIN rather than EACCES when a section of a file is already locked by another process, portable application programs should expect and test for either value.

Name

getmsg - get next message off a stream

Syntax

#include <stropts.h>

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

Description

The *getmsg* system call retrieves the contents of a message [see *intro*(S)] located at the *stream head* read queue from a STREAMS file, and places the contents into user-specified buffer(s). The message must contain either a data part, a control part, or both. The data and control parts of the message are placed into separate buffers, as described below. The semantics of each part is defined by the STREAMS module that generated the message.

The *fd* argument specifies a file descriptor referencing an open *stream. ctlptr* and *dataptr* each point to a *strbuf* structure which contains the following members:

```
int maxlen;     /* maximum buffer length */
int len;        /* length of data       */
char *buf;      /* ptr to buffer        */
```

where *buf* points to a buffer in which the data or control information is to be placed, and *maxlen* indicates the maximum number of bytes this buffer can hold. On return, *len* contains the number of bytes of data or control information actually received, or is 0 if there is a zero-length control or data part, or is -1 if no data or control information is present in the message. *flags* may be set to the values 0 or RS_HIPRI and is used as described below.

The *ctlptr* argument is used to hold the control part from the message and *dataptr* is used to hold the data part from the message. If *ctlptr* (or *dataptr*) is NULL or the *maxlen* field is -1, the control (or data) part of the message is not processed and is left on the *stream head* read queue, and *len* is set to -1. If the *maxlen* field is set to 0 and there is a zero-length control (or data) part, that zero-length part is removed from the read queue and *len* is set to 0. If the *maxlen* field is set to 0 and there are more than zero bytes of control (or data) information, that information is left on the read queue and *len* is set to 0. If the *maxlen* field in *ctlptr* or *dataptr* is less than, respectively, the control

or data part of the message, *maxlen* bytes are retrieved. In this case, the remainder of the message is left on the *stream head* read queue and a non-zero return value is provided, as described below under *DIAGNOSTICS*. If information is retrieved from a *priority* message, *flags* is set to RS_HIPRI on return.

By default, *getmsg* processes the first priority or non-priority message available on the *stream head* read queue. However, a user may choose to retrieve only priority messages by setting *flags* to RS_HIPRI. In this case, *getmsg* will only process the next message if it is a priority message.

If O_NDELAY has not been set, *getmsg* blocks until a message, of the type(s) specified by *flags* (priority or either), is available on the *stream head* read queue. If O_NDELAY has been set and a message of the specified type(s) is not present on the read queue, *getmsg* fails and sets *errno* to EAGAIN.

If a hangup occurs on the *stream* from which messages are to be retrieved, *getmsg* will continue to operate normally, as described above, until the *stream head* read queue is empty. Thereafter, it will return 0 in the *len* fields of *ctlptr* and *dataptr*.

The *getmsg* system call fails if one or more of the following is true:

[EAGAIN] The O_NDELAY flag is set, and no messages are available.

[EBADF] *fd* is not a valid file descriptor open for reading.

[EBADMSG] Queued message to be read is not valid for *getmsg*.

[EFAULT] *ctlptr*, *dataptr*, or *flags* points to a location outside the allocated address space.

[EINTR] A signal was caught during the *getmsg* system call.

[EINVAL] An illegal value was specified in *flags*, or the *stream* referenced by *fd* is linked under a multiplexer.

[ENOSTR] A *stream* is not associated with *fd*.

A *getmsg* can also fail if a STREAMS error message had been received at the *stream head* before the call to *getmsg*. The error returned is the value contained in the STREAMS error message.

See Also

intro(S), read(S), poll(S), putmsg(S), write(S).
STREAMS Primer
STREAMS Programmer's Guide

Diagnostics

Upon successful completion, a non-negative value is returned. A value of 0 indicates that a full message was read successfully. A return value of MORECTL indicates that more control information is waiting for retrieval. A return value of MOREDATA indicates that more data is waiting for retrieval. A return value of MORECTL|MOREDATA indicates that both types of information remain. Subsequent *getmsg* calls will retrieve the remainder of the message.

Name

ioctl - control device

Syntax

int ioctl (fildes, request, arg)
int fildes, request;

Description

The *ioctl* system call performs a variety of control functions on devices and STREAMS. For non-STREAMS files, the functions performed by this call are *device-specific* control functions. The arguments *request* and *arg* are passed to the file designated by *fildes* and are interpreted by the device driver. This control is infrequently used on non-STREAMS devices, with the basic input/output functions performed through the *read*(S) and *write*(S) system calls.

For STREAMS files, specific functions are performed by the *ioctl* call as described in *streamio*(STR).

fildes is an open file descriptor that refers to a device. *request* selects the control function to be performed and will depend on the device being addressed. *arg* represents additional information that is needed by this specific device to perform the requested function. The data type of *arg* depends upon the particular control request, but it is either an integer or a pointer to a device-specific data structure.

In addition to device-specific and STREAMS functions, generic functions are provided by more than one device driver, for example, the general terminal interface [see *termio*(M)].

The *ioctl* system call will fail for any type of file if one or more of the following is true:

[EBADF] *fildes* is not a valid open file descriptor.

[ENOTTY] *fildes* is not associated with a device driver that accepts control functions.

[EINTR] A signal was caught during the *ioctl* system call.

The *ioctl* system call will also fail if the device driver detects an error. In this case, the error is passed through *ioctl* without change to the caller. A particular driver might not have all of the following error cases. Other requests to device drivers will fail if one or more of the following is true:

[EFAULT] *request* requires a data transfer to or from a buffer pointed to by *arg*, but some part of the buffer is outside the process's allocated space.

[EINVAL] *request* or *arg* is not valid for this device.

[EIO] Some physical I/O error has occurred.

[ENXIO] The *request* and *arg* are valid for this device driver, but the service requested cannot be performed on this particular subdevice.

[ENOLINK] *fildes* is on a remote machine and the link to that machine is no longer active.

STREAMS errors are described in *streamio*(STR).

See Also

streamio(STR) in Appendix F of the *STREAMS Programmer's Guide.*
termio(M) in the *SCO XENIX User's Reference.*

Diagnostics

Upon successful completion, the value returned depends upon the device control function, but must be a non-negative integer. Otherwise, a value of -1 is returned, and *errno* is set to indicate the error.

Name

open - open for reading or writing

Syntax

#include <fcntl.h>
int open (path, oflag [, mode])
char *path;
int oflag, mode;

Description

path points to a path name naming a file. The *open* system call opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. For non-STREAMS [see *intro*(S)] files, *oflag* values are constructed by OR-ing flags from the following list (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY This flag may affect subsequent reads and writes [see *read*(S) and *write*(S)].

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NDELAY is set:

An *open* for reading-only will return without delay. An *open* for writing-only will return an error if no process currently has the file open for reading.

If O_NDELAY is clear:

An *open* for reading-only will block until a process opens the file for writing. An *open* for writing-only will block until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:

The open will return without waiting for carrier.

If O_NDELAY is clear:

The open will block until carrier is present.

O_APPEND If set, the file pointer will be set to the end of the file prior to each write.

O_SYNC When opening a regular file, this flag affects subsequent writes. If set, each *write*(S) will wait for both the file data and file status to be physically updated.

O_CREAT If the file exists, this flag has no effect. Otherwise, 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; and the low-order 12 bits of the file mode are set to the value of *mode*, modified as follows [see *creat*(S)]:

All bits set in the file mode creation mask of the process are cleared [see *umask*(S)].

The "save text image after execution bit" of the mode is cleared [see *chmod*(S)].

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

O_EXCL If O_EXCL and O_CREAT are set, *open* will fail if the file exists.

When opening a STREAMS file, *oflag* may be constructed from O_NDELAY 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 value of O_NDELAY affects the operation of STREAMS drivers and certain system calls [see *read*(S), *getmsg*(S), *putmsg*(S), and *write*(S)]. For drivers, the implementation of O_NDELAY is device-specific. Each STREAMS device driver may treat this option differently.

Certain flag values can be set following *open* as described in *fcntl*(S).

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across *exec* system calls [see *fcntl*(S)].

The named file is opened unless one or more of the following is true:

[EACCES] A component of the path prefix denies search permission.

[EACCES] *oflag* permission is denied for the named file.

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

[EEXIST] O_CREAT and O_EXCL are set, and the named file exists.

[EFAULT] *path* points outside the allocated address space of the process.

[EINTR] A signal was caught during the *open* system call.

[EIO] A hangup or error occurred during a STREAMS *open*.

[EISDIR] The named file is a directory and *oflag* is write or read/write.

[EMFILE] NOFILES file descriptors are currently open.

[EMULTIHOP] Components of *path* require hopping to multiple remote machines.

[ENFILE] The system file table is full.

[ENOENT] O_CREAT is not set and the named file does not exist.

[ENOLINK] *path* points to a remote machine, and the link to that machine is no longer active.

[ENOMEM] The system is unable to allocate a send descriptor.

[ENOSPC] O_CREAT and O_EXCL are set, and the file system is out of inodes.

[ENOSR] Unable to allocate a *stream.*

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

[ENXIO] The named file is a character special or block special file, and the device associated with this special file does not exist.

[ENXIO] O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.

[ENXIO] A STREAMS module or driver open routine failed.

[EROFS] The named file resides on a read-only file system and *oflag* is write or read/write.

[ETXTBSY] The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write.

See Also

chmod(S), close(S), creat(S), dup(S), fcntl(S), intro(S), lseek(S), read(S), getmsg(S), putmsg(S), umask(S), write(S).

Diagnostics

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

Name

poll - STREAMS input/output multiplexing

Syntax

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

int poll(fds, nfds, timeout)
struct pollfd fds[];
unsigned long nfds;
int timeout;
```

Description

The *poll* system call provides users with a mechanism for multiplexing input/output over a set of file descriptors that reference open *streams* [see *intro*(S)]. The *poll* system call identifies those *streams* on which a user can send or receive messages, or on which certain events have occurred. A user can receive messages using *read*(S) or *getmsg*(S) and can send messages using *write*(S) and *putmsg*(S). Certain *ioctl*(S) calls, such as I_RECVFD and I_SENDFD [see *streamio*(STR)], can also be used to receive and send messages.

fds specifies the file descriptors to be examined and the events of interest for each file descriptor. It is a pointer to an array with one element for each open file descriptor of interest. The array's elements are *pollfd* structures which contain the following members:

```
int fd;          /* file descriptor */
short events;    /* requested events */
short revents;   /* returned events */
```

where *fd* specifies an open file descriptor and *events* and *revents* are bitmasks constructed by or-ing any combination of the following event flags:

POLLIN — A non-priority or file descriptor passing message (see I_RECVFD) is present on the *stream head* read queue. This flag is set even if the message is of zero length. In *revents*, this flag is mutually exclusive with POLLPRI.

POLLPRI — A priority message is present on the *stream head* read queue. This flag is set even if the message is of zero length. In *revents*, this flag is mutually exclusive with POLLIN.

POLLOUT The first downstream write queue in the *stream* is not full. Priority control messages can be sent (see *putmsg*) at any time.

POLLERR An error message has arrived at the *stream head*. This flag is only valid in the *revents* bitmask; it is not used in the *events* field.

POLLHUP A hangup has occurred on the *stream*. This event and POLLOUT are mutually exclusive; a *stream* can never be writable if a hangup has occurred. However, this event and POLLIN or POLLPRI are not mutually exclusive. This flag is only valid in the *revents* bitmask; it is not used in the *events* field.

POLLNVAL The specified *fd* value does not belong to an open *stream*. This flag is only valid in the *revents* field; it is not used in the *events* field.

For each element of the array pointed to by *fds*, *poll* examines the given file descriptor for the event(s) specified in *events*. The number of file descriptors to be examined is specified by *nfds*. If *nfds* exceeds NOFILES, the system limit of open files [see *ulimit*(S)], *poll* will fail.

If the value *fd* is less than zero, *events* is ignored and *revents* is set to 0 in that entry on return from *poll*.

The results of the *poll* query are stored in the *revents* field in the *pollfd* structure. Bits are set in the *revents* bitmask to indicate which of the requested events are true. If none are true, none of the specified bits is set in *revents* when the *poll* call returns. The event flags POLLHUP, POLLERR, and POLLNVAL are always set in *revents* if the conditions they indicate are true; this occurs even though these flags were not present in *events*.

If none of the defined events have occurred on any selected file descriptor, *poll* waits at least *timeout* msec for an event to occur on any of the selected file descriptors. On a computer where millisecond timing accuracy is not available, *timeout* is rounded up to the nearest legal value available on that system. If the value *timeout* is 0, *poll* returns immediately. If the value of *timeout* is -1, *poll* blocks until a requested event occurs or until the call is interrupted. The *poll* system call is not affected by the O_NDELAY flag.

The *poll* system call fails if one or more of the following is true:

[EAGAIN] Allocation of internal data structures failed but request should be attempted again.

[EFAULT] Some argument points outside the allocated address space.

[EINTR] A signal was caught during the *poll* system call.

[EINVAL] The argument *nfds* is less than zero, or *nfds* is greater than NOFILES.

See Also

getmsg(S), intro(S), putmsg(S), read(S), write(S).
streamio(STR) in Appendix F of the *STREAMS Programmer's Guide*.
STREAMS Primer.

Diagnostics

Upon successful completion, a non-negative value is returned. A positive value indicates the total number of file descriptors that has been selected (that is, file descriptors for which the *revents* field is non-zero). A value of 0 indicates that the call timed out and no file descriptors have been selected. Upon failure, a value of -1 is returned, and *errno* is set to indicate the error.

Name

putmsg - send a message on a stream

Syntax

#include <stropts.h>

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

Description

The *putmsg* system call creates a message [see *intro*(S)] from user specified buffer(s) and sends the message to a STREAMS file. The message may contain either a data part, a control part or both. The data and control parts to be sent are distinguished by placement in separate buffers, as described below. The semantics of each part is defined by the STREAMS module that receives the message.

fd specifies a file descriptor referencing an open *stream. ctlptr* and *dataptr* each point to a *strbuf* structure which contains the following members:

```
int maxlen;     /* not used */
int len;        /* length of data */
char *buf;      /* ptr to buffer */
```

ctlptr points to the structure describing the control part, if any, to be included in the message. The *buf* field in the *strbuf* structure points to the buffer where the control information resides, and the *len* field indicates the number of bytes to be sent. The *maxlen* field is not used in *putmsg* [see *getmsg*(S)]. In a similar manner, *dataptr* specifies the data, if any, to be included in the message. *flags* may be set to the values 0 or RS_HIPRI and is used as described below.

To send the data part of a message, *dataptr* must be non-NULL and the *len* field of *dataptr* must have a value of 0 or greater. To send the control part of a message, the corresponding values must be set for *ctlptr*. No data (control) part will be sent if either *dataptr* (*ctlptr*) is NULL or the *len* field of *dataptr* (*ctlptr*) is set to -1.

If a control part is specified, and *flags* is set to RS_HIPRI, a *priority* message is sent. If *flags* is set to 0, a non-priority message is sent. If no control part is specified, and *flags* is set to RS_HIPRI, *putmsg* fails and sets *errno* to EINVAL. If no control part and no data part are specified, and *flags* is set to 0, no message is sent, and 0 is returned.

For non-priority messages, *putmsg* will block if the *stream* write queue is full due to internal flow control conditions. For priority messages, *putmsg* does not block on this condition. For non-priority messages, *putmsg* does not block when the write queue is full and O_NDELAY is set. Instead, it fails and sets *errno* to EAGAIN.

The *putmsg* system call also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the *stream*, regardless of priority or whether O_NDELAY has been specified. No partial message is sent.

The *putmsg* system call fails if one or more of the following is true:

[EAGAIN] A non-priority message was specified, the O_NDELAY flag is set, and the *stream* write queue is full due to internal flow control conditions.

[EAGAIN] Buffers could not be allocated for the message that was to be created.

[EBADF] *fd* is not a valid file descriptor open for writing.

[EFAULT] *ctlptr* or *dataptr* points outside the allocated address space.

[EINTR] A signal was caught during the *putmsg* system call.

[EINVAL] An undefined value was specified in *flags*, or *flags* is set to RS_HIPRI and no control part was supplied.

[EINVAL] The *stream* referenced by *fd* is linked below a multiplexer.

[ENOSTR] A *stream* is not associated with *fd*.

[ENXIO] A hangup condition was generated downstream for the specified *stream*.

[ERANGE] The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost *stream* module. This value is also returned if the control part of the message is larger than the maximum configured size of the control part of a message, or if the data part of a message is larger than the maximum configured size of the data part of a message.

A *putmsg* also fails if a STREAMS error message had been processed by the *stream* head before the call to *putmsg*. The error returned is the value contained in the STREAMS error message.

See Also

intro(S), read(S), getmsg(S), poll(S), write(S).
STREAMS Primer.
STREAMS Programmer's Guide.

Diagnostics

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

Name

read - read from file

Syntax

int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

Description

fildes is a file descriptor obtained from a *creat*(S), *open*(S), *dup*(S), *fcntl*(S), or *pipe*(S) system call.

The *read* system call attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*.

On devices capable of seeking, the *read* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *read*, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line [see *ioctl*(S) and *termio*(M)], or if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

A *read* from a STREAMS [see *intro*(S)] file can operate 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* request [see *streamio*(STR)], and can be tested with the I_GRDOPT *ioctl*. In byte-stream mode, *read* will retrieve data from the *stream* until it has retrieved *nbyte* bytes, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, *read* retrieves data until it has read *nbyte* bytes, or until it reaches a message boundary. If the *read* does not retrieve all the data in a message, the remaining data are replaced on the *stream*, and can be retrieved by the next *read* or *getmsg*(S) call. Message-discard mode also retrieves data until it has retrieved *nbyte* bytes, or it reaches a message boundary. However, unread data remaining in a message after the *read* returns are discarded and are not available for a subsequent *read* or *getmsg*.

When attempting to read from a regular file with mandatory file/record locking set [see *chmod*(S)], and there is a blocking (that is, owned by another process) write lock on the segment of the file to be read:

If O_NDELAY is set, the read will return a -1 and set errno to EAGAIN.

If O_NDELAY is clear, the read will sleep until the blocking record lock is removed.

When attempting to read from an empty pipe (or FIFO):

If O_NDELAY is set, the read will return a 0.

If O_NDELAY is clear, the read will block until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a tty that has no data currently available:

If O_NDELAY is set, the read will return a 0.

If O_NDELAY is clear, the read will block until data becomes available.

When attempting to read a file associated with a *stream* that has no data currently available:

If O_NDELAY is set, the read will return a -1 and set errno to EAGAIN.

If O_NDELAY is clear, the read will block until data becomes available.

When reading from a STREAMS file, handling of zero-byte messages is determined by the current read mode setting. In byte-stream mode, *read* accepts data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The *read* system call then returns the number of bytes read, and places the zero-byte message back on the *stream* to be retrieved by the next *read* or *getmsg*. In the two other modes, a zero-byte message returns a value of 0 and the message is removed from the *stream*. When a zero-byte message is read as the first message on a *stream*, a value of 0 is returned regardless of the read mode.

A *read* from a STREAMS file can only process data messages. It cannot process any type of protocol message and will fail if a protocol message is encountered at the *stream head*.

The *read* system call will fail if one or more of the following are true:

[EAGAIN] Mandatory file/record locking was set, O_NDELAY was set, and there was a blocking record lock.

[EAGAIN] Total amount of system memory available when reading via raw I/O is temporarily insufficient.

[EAGAIN] No message waiting to be read on a *stream* and O_NDELAY flag set.

[EBADF] *fildes* 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 situation to occur.

[EFAULT] *buf* points outside the allocated address space.

[EINTR] A signal was caught during the *read* system call.

[EIO] A physical I/O error has occurred.

[ENXIO] The device associated with the file-descriptor is a block-special or character-special file, and the value of the file-pointer is out of range.

[EINVAL] Attempted to read from a *stream* linked to a multiplexer.

[ENOLCK] The system record lock table was full, so the read could not go to sleep until the blocking record lock was removed.

[ENOLINK] *fildes* is on a remote machine and the link to that machine is no longer active.

A *read* from a STREAMS file will also fail if an error message is received at the *stream head.* In this case, *errno* is set to the value returned in the error message. If a hangup occurs on the *stream* being read, *read* will continue to operate normally until the *stream head* read queue is empty. Thereafter, it will return 0.

See Also

creat(S), dup(S), fcntl(S), ioctl(S), intro(2), open(S), pipe(S), getmsg(S).
streamio(STR), in Appendix F of the *STREAMS Programmer's Guide.*
termio(M) in the *XENIX Reference.*

Diagnostics

Upon successful completion a non-negative integer is returned indicating the number of bytes actually read. Otherwise, a -1 is returned, and *errno* is set to indicate the error.

Name

signal - specify what to do upon receipt of a signal

Syntax

#include <signal.h>

void (*signal (sig, func))( )
int sig;
void (*func)( );

Description

The *signal* system call allows the calling process to choose one of three ways in which it is possible to handle the receipt of a specific signal. *sig* specifies the signal and *func* specifies the choice.

sig can be assigned any one of the following except **SIGKILL**:

SIGHUP	01	hangup
SIGINT	02	interrupt
SIGQUIT	03[1]	quit
SIGILL	04[1]	illegal instruction (not reset when caught)
SIGTRAP	05[1]	trace trap (not reset when caught)
SIGIOT	06[1]	IOT instruction
SIGABRT	06	used by abort, replaces SIG10T
SIGEMT	07[1]	EMT instruction
SIGFPE	08[1]	floating point exception
SIGKILL	09	kill (cannot be caught or ignored)
SIGBUS	10[1]	bus error
SIGSEGV	11[1]	segmentation violation
SIGSYS	12[1]	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTERM	15	software termination signal
SIGUSR1	16	user-defined signal 1
SIGUSR2	17	user-defined signal 2
SIGCLD	18[2]	death of a child
SIGPWR	19[2]	power fail
SIGPOLL	22[3]	selectable event pending

func is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. **SIG_DFL**, and **SIG_IGN**, are defined in the include file **<signal.h>**. Each is a macro that expands to a constant expression of type pointer to function returning *void*, and has a unique value that matches no declarable function.

The actions prescribed by the values of *func* are as follows:

SIG_DFL —terminate process upon receipt of a signal
Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(S). See NOTE [1] below.

SIG_IGN —ignore signal
The signal *sig* is to be ignored.

Note: the signal **SIGKILL** cannot be ignored.

function address —catch signal
Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function pointed to by *func*. The signal number *sig* will be passed as the only argument to the signal-catching function. Additional arguments are passed to the signal-catching function for hardware-generated signals. Before entering the signal-catching function, the value of *func* for the caught signal will be set to **SIG_DFL** unless the signal is **SIGILL**, **SIGTRAP**, or **SIGPWR**.

Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted.

When a signal that is to be caught occurs during a *read*(S), a *write*(S), an *open*(S), or an *ioctl*(S) system call on a slow device (like a terminal; but not a file), during a *pause*(S) system call, or during a *wait*(S) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed. Then the interrupted system call may return a -1 to the calling process with *errno* set to EINTR.

The *signal* system call will not catch an invalid function argument, *func*, and results are undefined when an attempt is made to execute the function at the bad address.

Note: The signal **SIGKILL** cannot be caught.

A call to *signal* cancels a pending signal *sig* except for a pending **SIGKILL** signal.

The *signal* system call will fail if *sig* is an illegal signal number, including **SIGKILL**. [EINVAL]

See Also

intro(S), kill(S), pause(S), ptrace(S), wait(S), setjmp(S), sigset(S).

Diagnostics

Upon successful completion, *signal* returns the previous value of *func* for the specified signal *sig*. Otherwise, a value of SIG_ERR is returned and *errno* is set to indicate the error. SIG_ERR is defined in the include file *signal.h*.

Notes

[1] If **SIG_DFL** is assigned for these signals, in addition to the process being terminated, a "core image" will be constructed in the current working directory of the process, if the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

- a mode of 0666 modified by the file creation mask [see *umask*(S)]
- a file owner ID that is the same as the effective user ID of the receiving process
- a file group ID that is the same as the effective group ID of the receiving process.

[2] For the signals **SIGCLD** and **SIGPWR**, *func* is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. The actions prescribed by these values are:

SIG_DFL —ignore signal
The signal is to be ignored.

SIG_IGN —ignore signal
The signal is to be ignored. Also, if *sig* is **SIGCLD**, the calling process's child processes will not create zombie processes when they terminate [see *exit*(S)].

function address —catch signal
If the signal is **SIGPWR**, the action to be taken is the same as that described above for *func* equal to *function address*. The same is true if the signal is

SIGCLD with one exception: while the process is executing the signal-catching function, any received **SIGCLD** signals will be ignored. (This is the default action.)

In addition, **SIGCLD** affects the *wait* and *exit* system calls as follows:

wait If the *func* value of **SIGCLD** is set to **SIG_IGN** and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value of -1 with *errno* set to ECHILD.

exit If in the exiting process's parent process the *func* value of **SIGCLD** is set to **SIG_IGN**, the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the preceding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

[3] **SIGPOLL** is issued when a file descriptor corresponding to a STREAMS [see *intro*(S)] file has a "selectable" event pending. A process must specifically request that this signal be sent using the I_SETSIG *ioctl* call. Otherwise, the process will never receive **SIGPOLL**.

Name

sigset, sighold, sigrelse, sigignore, sigpause - signal management

Syntax

#include <signal.h>

void (*sigset (sig, func))( )
int sig;
void (*func)( );

int sighold (sig)
int sig;

int sigrelse (sig)
int sig;

int sigignore (sig)
int sig;

int sigpause (sig)
int sig;

Description

These functions provide signal management for application processes. The *sigset* system call specifies the system signal action to be taken upon receipt of signal *sig*. This action is either calling a process signal-catching handler *func* or performing a system-defined action.

Sig can be assigned any one of the following values except SIGKILL. Machine- or implementation-dependent signals are not included (see *NOTES* below). Each value of *sig* is a macro, defined in **<signal.h>**, that expands to an integer constant expression.

SIGHUP	hangup
SIGINT	interrupt
SIGQUIT*	quit
SIGILL*	illegal instruction (not held when caught)
SIGTRAP*	trace trap (not held when caught)
SIGABRT*	abort
SIGFPE*	floating point exception
SIGKILL	kill (cannot be caught or ignored)
SIGSYS*	bad argument to system call
SIGPIPE	write on a pipe with no one to read it
SIGALRM	alarm clock
SIGTERM	software termination signal
SIGUSR1	user-defined signal 1
SIGUSR2	user-defined signal 2
SIGCLD	death of a child (see *WARNING* below)
SIGPWR	power fail (see *WARNING* below)

SIGPOLL selectable event pending (see *NOTES* below)

See below under SIG_DFL regarding asterisks (*) in the above list.

The following values for the system-defined actions of *func* are also defined in **<signal.h>**. Each is a macro that expands to a constant expression of type pointer to function returning *void* and has a unique value that matches no declarable function.

SIG_DFL—default system action
Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in *exit*(S). In addition a "core image" will be made in the current working directory of the receiving process if *sig* is one for which an asterisk appears in the above list and the following conditions are met:

The effective user ID and the real user ID of the receiving process are equal.

An ordinary file named **core** exists and is writable or can be created. If the file must be created, it will have the following properties:

a mode of 0666 modified by the file creation mask [see *umask*(S)]

a file owner ID that is the same as the effective user ID of the receiving process

a file group ID that is the same as the effective group ID of the receiving process.

SIG_IGN—ignore signal
Any pending signal *sig* is discarded and the system signal action is set to ignore future occurrences of this signal type.

SIG_HOLD—hold signal
The signal *sig* is to be held upon receipt. Any pending signal of this type remains held. Only one signal of each type is held.

Otherwise, *func* must be a pointer to a function, the signal-catching handler, that is to be called when signal *sig* occurs. In this case, *sigset* specifies that the process will call this function upon receipt of signal *sig*. Any pending signal of this type is released. This handler address is retained across calls to the other signal management functions listed here.

When a signal occurs, the signal number *sig* will be passed as the only argument to the signal-catching handler. Before calling the signal-catching handler, the system signal action will be set to SIG_HOLD.

During normal return from the signal-catching handler, the system signal action is restored to *func* and any held signal of this type released. If a non-local goto (*longjmp*) is taken, then *sigrelse* must be called to restore the system signal action and release any held signal of this type.

In general, upon return from the signal-catching handler, the receiving process will resume execution at the point it was interrupted. However, when a signal is caught during a *read*(S), a *write*(S), an *open*(S), or an *ioctl* (S) system call during a *sigpause* system call, or during a *wait*(S) system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal-catching handler will be executed. Then the interrupted system call may return a -1 to the calling process with *errno* set to EINTR.

sighold and *sigrelse* are used to establish critical regions of code. *sighold* is analogous to raising the priority level and deferring or holding a signal until the priority is lowered by *sigrelse*. *sigrelse* restores the system signal action to that specified previously by *sigset*.

sigignore sets the action for signal *sig* to SIG_IGN (see above).

sigpause suspends the calling process until it receives a signal, the same as *pause*(S). However, if the signal *sig* had been received and held, it is released and the system signal action taken. This system call is useful for testing variables that are changed on the occurrence of a signal. The correct usage is to use *sighold* to block the signal first, then test the variables. If they have not changed, then call *sigpause* to wait for the signal. *sigset* will fail if one or more of the following is true:

[EINVAL] *sig* is an illegal signal number (including **SIGKILL**) or the default handling of *sig* cannot be changed.

[EINTR] A signal was caught during the system call *sigpause*.

See Also

kill(S), pause(S), signal(S), wait(S), setjmp(S).

Diagnostics

Upon successful completion, *sigset* returns the previous value of the system signal action for the specified signal *sig*. Otherwise, a value of SIG_ERR is returned and *errno* is set to indicate the error. SIG_ERR is defined in **<signal.h>**.

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

Notes

SIGPOLL is issued when a file descriptor corresponding to a STREAMS [see *intro*(S)] file has a "selectable" event pending. A process must specifically request that this signal be sent using the I_SETSIG *ioctl*(S) call [see *streamio*(STR)]. Otherwise, the process will never receive **SIGPOLL**.

For portability, applications should use only the symbolic names of signals rather than their values and use only the set of signals defined here. The action for the signal **SIGKILL** cannot be changed from the default system action.

Specific implementations may have other implementation-de fined signals. Also, additional implementation-de fined arguments may be passed to the signal-catching handler for hardware-generated signals. For certain hardware-generated signals, it may not be possible to resume execution at the point of interruption.

The signal type **SIGSEGV** is reserved for the condition that occurs on an invalid access to a data object. If an implementation can detect this condition, this signal type should be used.

The other signal management functions, *signal*(S) and *pause*(S), should not be used in conjunction with these routines for a particular signal type.

WARNING

Two signals that behave differently from the signals described above exist in this release of the system:

SIGCLD	death of a child (reset when caught)
SIGPWR	power fail (not reset when caught)

For these signals, *func* is assigned one of three values: SIG_DFL, SIG_IGN, or a *function address*. The actions prescribed by these values are as follows:

SIG_DFL-ignore signal
The signal is to be ignored.

SIG_IGN-ignore signal
The signal is to be ignored. Also, if *sig* is **SIGCLD**, the calling process's child processes will not create zombie processes when they terminate [see *exit*(S)].

s*function address*-catch signal
If the signal is **SIGPWR**, the action to be taken is the same as that described above for *func* equal to *function address*. The same is true if the signal is **SIGCLD** with one exception: while the process is executing the signal-catching function, any received **SIGCLD** signals will be ignored. (This is the default action.)

The **SIGCLD** affects two other system calls [*wait*(S), and *exit*(S)] in the following ways:

wait If the *func* value of **SIGCLD** is set to SIG_IGN and a *wait* is executed, the *wait* will block until all of the calling process's child processes terminate; it will then return a value of -1 with *errno* set to ECHILD.

exit If in the exiting process's parent process the *func* value of **SIGCLD** is set to SIG_IGN, the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the preceding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set **SIGCLD** to be caught.

Name

write - write on a file

Syntax

int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;

Description

fildes is a file descriptor obtained from a *creat*(S), *open*(S), *dup*(S), *fcntl*(S), or *pipe*(S) system call.

The *write* system call attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write.

For regular files, if the O_SYNC flag of the file status flags is set, the write will not return until both the file data and file status have been physically updated. This function is for special applications that require extra reliability at the cost of performance. For block special files, if O_SYNC is set, the write will not return until the data has been physically updated.

A write to a regular file will be blocked if mandatory file/record locking is set [see *chmod*(S)], and there is a record lock owned by another process on the segment of the file to be written. If O_NDELAY is not set, the write will sleep until the blocking record lock is removed.

For STREAMS [see *intro*(S)] files, 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. Unless the user pushes [see I_PUSH in *streamio*(STR)] the topmost module, these values cannot be set or tested from user level. If *nbyte* falls within the packet size range, *nbyte* bytes will be written. If *nbyte* does not fall within the range and

the minimum packet size value is zero, *write* will break the buffer into maximum packet size segments prior to sending the data downstream (the last segment may contain less than the maximum packet size). If *nbyte* does not fall within the range and the minimum value is non-zero, *write* will fail with *errno* set to ERANGE. Writing a zero-length buffer (*nbyte* is zero) sends zero bytes with zero returned.

For STREAMS files, if O_NDELAY is not set and the *stream* cannot accept data (the *stream* write queue is full due to internal flow control conditions), *write* will block until data can be accepted. O_NDELAY will prevent a process from blocking due to flow control conditions. If O_NDELAY is set and the *stream* cannot accept data, *write* will fail. If O_NDELAY is set and part of the buffer has been written when a condition in which the *stream* cannot accept additional data occurs, *write* will terminate and return the number of bytes written.

The *write* system call will fail and the file pointer will remain unchanged if one or more of the following is true:

[EAGAIN] Mandatory file/record locking was set, O_NDELAY was set, and there was a blocking record lock.

[EAGAIN] Total amount of system memory available when reading via raw I/O is temporarily insufficient.

[EAGAIN] Attempt to write to a *stream* that cannot accept data with the O_NDELAY flag set.

[EBADF] *fildes* 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.

[EFAULT] *buf* points outside the process's allocated address space.

[EFBIG] An attempt was made to write a file that exceeds the process's file size limit or the maximum file size [see *ulimit*(S)].

[EINTR] A signal was caught during the *write* system call.

[EINVAL] Attempt to write to a *stream* linked below a multiplexer.

[ENOLCK] The system record lock table was full, so the write could not go to sleep until the blocking record lock was removed.

[ENOLINK] *fildes* is on a remote machine 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.

[ENXIO] A hangup occurred on the *stream* being written to.

[EPIPE and SIGPIPE signal]
An attempt is made to write to a pipe that is not open for reading by any process.

[ERANGE] Attempt to write to a *stream* with *nbyte* outside specified minimum and maximum write range, and the minimum value is non-zero.

[EIO] A physical I/O error has occurred.

If a *write* requests that more bytes be written than there is room for (for example, the *ulimit* [see *ulimit*(S)] or the physical end of a medium), 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 will return 20. The next write of a non-zero number of bytes will give a failure return (except as noted below).

If the file being written is a pipe (or FIFO) and the O_NDELAY flag of the file flag word is set, then write to a full pipe (or FIFO) will return a count of 0. Otherwise (O_NDELAY clear), writes to a full pipe (or FIFO) will block until space becomes available.

A write to a STREAMS file can fail if an error message has been received at the stream head. In this case, *errno* is set to the value included in the error message.

See Also

creat(S), dup(S), fcntl(S), intro(S), lseek(S), open(S), pipe(S), ulimit(S).

Diagnostics

Upon successful completion the number of bytes actually written is returned. Otherwise, -1 is returned, and *errno* is set to indicate the error.