NAME

accept – accept a connection on a socket

SYNOPSIS

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

int accept(int *s*, **struct sockaddr** **addr*, **int** **addrlen*);

DESCRIPTION

The argument *s* is a socket that has been created with **socket**(3N) and bound to an address with **bind**(3N), and that is listening for connections after a call to **listen**(3N). The **accept**( ) function extracts the first connection on the queue of pending connections, creates a new socket with the properties of *s*, and allocates a new file descriptor, *ns*, for the socket. If no pending connections are present on the queue and the socket is not marked as non-blocking, **accept**( ) blocks the caller until a connection is present. If the socket is marked as non-blocking and no pending connections are present on the queue, **accept**( ) returns an error as described below. The **accept**( ) function uses the **netconfig**(4) file to determine the STREAMS device file name associated with *s*. This is the device on which the connect indication will be accepted. The accepted socket, *ns*, is used to read and write data to and from the socket that connected to *ns*; it is not used to accept more connections. The original socket (*s*) remains open for accepting further connections.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity as it is known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication occurs.

The argument *addrlen* is a value-result parameter. Initially, it contains the amount of space pointed to by *addr*; on return it contains the length in bytes of the address returned.

The **accept**( ) function is used with connection-based socket types, currently with **SOCK_STREAM**.

It is possible to **select**(3C) or **poll**(2) a socket for the purpose of an **accept**( ) by selecting or polling it for a read. However, this will only indicate when a connect indication is pending; it is still necessary to call **accept**( ).

RETURN VALUE

On success, these system calls return a nonnegative integer that is a file descriptor for the accepted socket. On error, −1 is returned, and *errno* is set appropriately.

ERRORS

accept( ) will fail if:

EBADF	The descriptor is invalid.
EINTR	The accept attempt was interrupted by the delivery of a signal.
EMFILE	The per-process descriptor table is full.
ENODEV	The protocol family and type corresponding to *s* could not be found in the **netconfig** file.
ENOMEM	There was insufficient user memory available to complete the operation.
EPROTO	A protocol error has occurred; for example, the STREAMS protocol stack has not been initialized or the connection has already been released.
EWOULDBLOCK	The socket is marked as non-blocking and no connections are present to be accepted.

SEE ALSO

poll(2), **bind**(3N), **connect**(3N), **listen**(3N), **select**(3C), **socket**(3N), **netconfig**(4), **attributes**(5), **socket**(5)

NAME

bind – bind a name to a socket

SYNOPSIS

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

int bind(int *s*, **const struct sockaddr** **name*, **int** *namelen*);

DESCRIPTION

bind( ) assigns a name to an unnamed socket *s* . When a socket is created with **socket**(3N), it exists in a name space (address family) but has no name assigned. **bind**( ) requests that the name pointed to by *name* be assigned to the socket.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set appropriately.

ERRORS

The **bind**( ) call will fail if:

EACCES	The requested address is protected and the current user has inadequate permission to access it.
EADDRINUSE	The specified address is already in use.
EADDRNOTAVAIL	The specified address is not available on the local machine.
EBADF	*s* is not a valid descriptor.
EINVAL	*namelen* is not the size of a valid address for the specified address family.
EINVAL	The socket is already bound to an address.
ENOSR	There were insufficient STREAMS resources for the operation to complete.
ENOTSOCK	*s* is a descriptor for a file, not a socket.

The following errors are specific to binding names in the UNIX domain:

EACCES	Search permission is denied for a component of the path prefix of the pathname in *name*.
EIO	An I/O error occurred while making the directory entry or allocating the inode.
EISDIR	A null pathname was specified.
ELOOP	Too many symbolic links were encountered in translating the pathname in *name*.
ENOENT	A component of the path prefix of the pathname in *name* does not exist.
ENOTDIR	A component of the path prefix of the pathname in *name* is not a directory.
EROFS	The inode would reside on a read-only file system.

SEE ALSO

unlink(2), **socket**(3N), **attributes**(5), **socket**(5)

NOTES

Binding a name in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed (using **unlink**(2)).

The rules used in name binding vary between communication domains.

NAME

fdopen – associate a stream with a file descriptor

SYNOPSIS

#include <stdio.h>

FILE *fdopen(int *fildes*, **const char *** *mode*);

DESCRIPTION

The **fdopen**( ) function associates a stream with a file descriptor *fildes*, whose value must be less than 255.

The *mode* argument is a character string having one of the following values:

r or **rb**	open a file for reading
w or **wb**	open a file for writing
a or **ab**	open a file for writing at end of file
r+ or **rb+** or **r+b**	open a file for update (reading and writing)
w+ or **wb+** or **w+b**	open a file for update (reading and writing)
a+ or **ab+** or **a+b**	open a file for update (reading and writing) at end of file

The meaning of these flags is exactly as specified in **fopen**(3S), except that modes beginning with **w** do not cause truncation of the file.

The mode of the stream must be allowed by the file access mode of the open file. The file position indicator associated with the new stream is set to the position indicated by the file offset associated with the file descriptor.

fdopen( ) will preserve the offset maximum previously set for the open file description corresponding to *fildes*.

The error and end-of-file indicators for the stream are cleared. The **fdopen**( ) function may cause the **st_atime** field of the underlying file to be marked for update.

RETURN VALUES

Upon successful completion, **fdopen**( ) returns a pointer to a stream. Otherwise, a null pointer is returned and **errno** is set to indicate the error.

fdopen( ) may fail and not set **errno** if there are no free **stdio** streams.

ERRORS

The **fdopen**( ) function may fail if:

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

EINVAL The *mode* argument is not a valid mode.

EMFILE **FOPEN_MAX** streams are currently open in the calling process.

EMFILE **STREAM_MAX** streams are currently open in the calling process.

ENOMEM Insufficient space to allocate a buffer.

USAGE

STREAM_MAX is the number of streams that one process can have open at one time. If defined, it has the same value as **FOPEN_MAX**.

File descriptors are obtained from calls like **open**(2), **dup**(2), **creat**(2) or **pipe**(2), which open files but do not return streams. Streams are necessary input for almost all of the Section 3S library routines.

SEE ALSO

creat(2), **dup**(2), **open**(2), **pipe**(2), **fclose**(3S), **fopen**(3S), **attributes**(5)

NAME

clearerr, feof, ferror, fileno – check and reset stream status

SYNOPSIS

#include <stdio.h>

void clearerr(FILE * *stream*);
int feof(FILE * *stream*);
int ferror(FILE * *stream*);
int fileno(FILE * *stream*);

DESCRIPTION

The function **clearerr**() clears the end-of-file and error indicators for the stream pointed to by *stream*.

The function **feof**() tests the end-of-file indicator for the stream pointed to by *stream*, returning non-zero if it is set. The end-of-file indicator can only be cleared by the function **clearerr**().

The function **ferror**() tests the error indicator for the stream pointed to by *stream*, returning non-zero if it is set. The error indicator can only be reset by the **clearerr**() function.

The function **fileno**() examines the argument *stream* and returns its integer descriptor.

For non-locking counterparts, see **unlocked_stdio**(3).

ERRORS

These functions should not fail and do not set the external variable *errno*. (However, in case **fileno**() detects that its argument is not a valid stream, it must return −1 and set *errno* to **EBADF**.)

CONFORMING TO

The functions **clearerr**(), **feof**(), and **ferror**() conform to C89 and C99.

SEE ALSO

open(2), **fdopen**(3), **stdio**(3), **unlocked_stdio**(3)

NAME

fflush – flush a stream

SYNOPSIS

#include <stdio.h>

int fflush(FILE ***stream*****);**

DESCRIPTION

For output streams, **fflush**() forces a write of all user-space buffered data for the given output or update *stream* via the stream's underlying write function.

For input streams associated with seekable files (e.g., disk files, but not pipes or terminals), **fflush**() discards any buffered data that has been fetched from the underlying file, but has not been consumed by the application.

The open status of the stream is unaffected.

If the *stream* argument is NULL, **fflush**() flushes *all* open output streams.

For a nonlocking counterpart, see **unlocked_stdio**(3).

RETURN VALUE

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

ERRORS

EBADF

stream is not an open stream, or is not open for writing.

The function **fflush**() may also fail and set *errno* for any of the errors specified for **write**(2).

SEE ALSO

fsync(2), **sync**(2), **write**(2), **fclose**(3), **fileno**(3), **fopen**(3), **setbuf**(3), **unlocked_stdio**(3)

NAME

fopen, fdopen, fileno – stream open functions

SYNOPSIS

#include <stdio.h>

FILE *fopen(const char * *path*, **const char *** *mode*);
FILE *fdopen(int *fildes*, **const char *** *mode*);
int fileno(FILE * *stream*);
int fclose(FILE * *stream*);

DESCRIPTION

The **fopen** function opens the file whose name is the string pointed to by *path* and associates a stream with it.

The argument *mode* points to a string beginning with one of the following sequences (Additional characters may follow these sequences.):

r Open text file for reading. The stream is positioned at the beginning of the file.

r+ Open for reading and writing. The stream is positioned at the beginning of the file.

w Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.

w+ Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.

a Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

a+ Open for reading and appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

The **fdopen** function associates a stream with the existing file descriptor, *fildes*. The *mode* of the stream (one of the values "r", "r+", "w", "w+", "a", "a+") must be compatible with the mode of the file descriptor. The file position indicator of the new stream is set to that belonging to *fildes*, and the error and end-of-file indicators are cleared. Modes "w" or "w+" do not cause truncation of the file. The file descriptor is not dup'ed, and will be closed when the stream created by **fdopen** is closed. The result of applying **fdopen** to a shared memory object is undefined.

The function **fileno**() examines the argument *stream* and returns its integer descriptor.

The **fclose**() function flushes the stream pointed to by *stream* (writing any buffered output data using **fflush**(3)) and closes the underlying file descriptor.

RETURN VALUE

Upon successful completion **fopen**, **fdopen** and **freopen** return a **FILE** pointer. Otherwise, **NULL** is returned and the global variable *errno* is set to indicate the error. Upon successful completion of **fclose**, 0 is returned. Otherwise, **EOF** is returned and *errno* is set to indicate the error.

ERRORS

EINVAL

The *mode* provided to **fopen**, **fdopen**, or **freopen** was invalid.

EBADF

The file descriptor underlying *stream* passed to **fclose** is not valid.

The **fopen**, **fdopen** and **freopen** functions may also fail and set *errno* for any of the errors specified for the routine **malloc**(3).

The **fopen** function may also fail and set *errno* for any of the errors specified for the routine **open**(2).

The **fdopen** function may also fail and set *errno* for any of the errors specified for the routine **fcntl**(2).

NAME

fread, fwrite – binary stream input/output

SYNOPSIS

#include <stdio.h>

size_t fread(void * *ptr***, size_t** *size***, size_t** *nmemb***, FILE *** *stream***);**

size_t fwrite(const void * *ptr***, size_t** *size***, size_t** *nmemb***,**
FILE * *stream***);**

DESCRIPTION

The function **fread**() reads *nmemb* elements of data, each *size* bytes long, from the stream pointed to by *stream*, storing them at the location given by *ptr*.

The function **fwrite**() writes *nmemb* elements of data, each *size* bytes long, to the stream pointed to by *stream*, obtaining them from the location given by *ptr*.

For non-locking counterparts, see **unlocked_stdio**(3).

RETURN VALUE

fread() and **fwrite**() return the number of items successfully read or written (i.e., not the number of characters). If an error occurs, or the end-of-file is reached, the return value is a short item count (or zero).

fread() does not distinguish between end-of-file and error, and callers must use **feof**(3) and **ferror**(3) to determine which occurred.

CONFORMING TO

C89, POSIX.1-2001.

SEE ALSO

read(2), **write**(2), **feof**(3), **ferror**(3), **unlocked_stdio**(3)

NAME

fgetc, fgets, getc, getchar, fputc, fputs, putc, putchar – input and output of characters and strings

SYNOPSIS

#include <stdio.h>

int fgetc(FILE * *stream***);**
char *fgets(char * *s***, int** *size***, FILE *** *stream***);**
int getc(FILE * *stream***);**
int getchar(void);
int fputc(int *c***, FILE *** *stream***);**
int fputs(const char * *s***, FILE *** *stream***);**
int putc(int *c***, FILE *** *stream***);**
int putchar(int *c***);**

DESCRIPTION

fgetc() reads the next character from *stream* and returns it as an *unsigned char* cast to an *int*, or **EOF** on end of file or error.

getc() is equivalent to **fgetc**() except that it may be implemented as a macro which evaluates *stream* more than once.

getchar() is equivalent to **getc**(*stdin*).

fgets() reads in at most one less than *size* characters from *stream* and stores them into the buffer pointed to by *s*. Reading stops after an **EOF** or a newline. If a newline is read, it is stored into the buffer. A **'\0'** is stored after the last character in the buffer.

fputc() writes the character *c*, cast to an *unsigned char*, to *stream*.

fputs() writes the string *s* to *stream*, without its terminating null byte ('\0').

putc() is equivalent to **fputc**() except that it may be implemented as a macro which evaluates *stream* more than once.

putchar(*c***);** is equivalent to **putc(***c***,** *stdout***).**

Calls to the functions described here can be mixed with each other and with calls to other output functions from the *stdio* library for the same output stream.

RETURN VALUE

fgetc(), **getc**() and **getchar**() return the character read as an *unsigned char* cast to an *int* or **EOF** on end of file or error.

fgets() returns *s* on success, and NULL on error or when end of file occurs while no characters have been read. **fputc**(), **putc**() and **putchar**() return the character written as an *unsigned char* cast to an *int* or **EOF** on error.

fputs() returns a nonnegative number on success, or **EOF** on error.

SEE ALSO

read(2), **write**(2), **ferror**(3), **fgetwc**(3), **fgetws**(3), **fopen**(3), **fread**(3), **fseek**(3), **getline**(3), **getwchar**(3), **scanf**(3), **ungetwc**(3), **write**(2), **ferror**(3), **fopen**(3), **fputwc**(3), **fputws**(3), **fseek**(3), **fwrite**(3), **gets**(3), **putwchar**(3), **scanf**(3), **unlocked_stdio**(3)

NAME

ipv6, AF_INET6 – Linux IPv6 protocol implementation

SYNOPSIS

```
#include <sys/socket.h>
#include <netinet/in.h>

tcp6_socket = socket(AF_INET6, SOCK_STREAM, 0);
raw6_socket = socket(AF_INET6, SOCK_RAW, protocol);
udp6_socket = socket(AF_INET6, SOCK_DGRAM, protocol);
```

DESCRIPTION

Linux 2.2 optionally implements the Internet Protocol, version 6. This man page contains a description of the IPv6 basic API as implemented by the Linux kernel and glibc 2.1. The interface is based on the BSD sockets interface; see **socket**(7).

The IPv6 API aims to be mostly compatible with the **ip**(7) v4 API. Only differences are described in this man page.

To bind an **AF_INET6** socket to any process the local address should be copied from the *in6addr_any* variable which has *in6_addr* type. In static initializations **IN6ADDR_ANY_INIT** may also be used, which expands to a constant expression. Both of them are in network order.

IPv4 connections can be handled with the v6 API by using the v4-mapped-on-v6 address type; thus a program only needs only to support this API type to support both protocols. This is handled transparently by the address handling functions in libc.

IPv4 and IPv6 share the local port space. When you get an IPv4 connection or packet to a IPv6 socket its source address will be mapped to v6 and it will be mapped to v6.

Address Format

```
struct sockaddr_in6 {
    uint16_t        sin6_family;    /* AF_INET6 */
    uint16_t        sin6_port;      /* port number */
    uint32_t        sin6_flowinfo;  /* IPv6 flow information */
    struct in6_addr sin6_addr;      /* IPv6 address */
    uint32_t        sin6_scope_id;  /* Scope ID (new in 2.4) */
};

struct in6_addr {
    unsigned char   s6_addr[16];    /* IPv6 address */
};
```

sin6_family is always set to **AF_INET6**; *sin6_port* is the protocol port (see *sin_port* in **ip**(7)); *sin6_flowinfo* is the IPv6 flow identifier; *sin6_addr* is the 128-bit IPv6 address. *sin6_scope_id* is an ID of depending of on the scope of the address. It is new in Linux 2.4. Linux only supports it for link scope addresses, in that case *sin6_scope_id* contains the interface index (see **netdevice**(7))

RETURN VALUES

−1 is returned if an error occurs. Otherwise the return value is a descriptor referencing the socket.

NOTES

The *sockaddr_in6* structure is bigger than the generic *sockaddr*. Programs that assume that all address types can be stored safely in a *struct sockaddr* need to be changed to use *struct sockaddr_storage* for that instead.

SEE ALSO

cmsg(3), **ip**(7)

NAME

listen – listen for connections on a socket

SYNOPSIS

```
#include <sys/types.h>      /* See NOTES */
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

DESCRIPTION

listen() marks the socket referred to by *sockfd* as a passive socket, that is, as a socket that will be used to accept incoming connection requests using **accept**(2).

The *sockfd* argument is a file descriptor that refers to a socket of type **SOCK_STREAM** or **SOCK_SEQPACKET**.

The *backlog* argument defines the maximum length to which the queue of pending connections for *sockfd* may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of **ECONNREFUSED** or, if the underlying protocol supports retransmission, the request may be ignored so that a later reattempt at connection succeeds.

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set appropriately.

ERRORS

EADDRINUSE
: Another socket is already listening on the same port.

EBADF
: The argument *sockfd* is not a valid descriptor.

ENOTSOCK
: The argument *sockfd* is not a socket.

NOTES

To accept connections, the following steps are performed:

1. A socket is created with **socket**(2).
2. The socket is bound to a local address using **bind**(2), so that other sockets may be **connect**(2)ed to it.
3. A willingness to accept incoming connections and a queue limit for incoming connections are specified with **listen**().
4. Connections are accepted with **accept**(2).

If the *backlog* argument is greater than the value in */proc/sys/net/core/somaxconn*, then it is silently truncated to that value; the default value in this file is 128.

EXAMPLE

See **bind**(2).

SEE ALSO

accept(2), **bind**(2), **connect**(2), **socket**(2), **socket**(7)

NAME

calloc, malloc, free, realloc – Allocate and free dynamic memory

SYNOPSIS

#include <stdlib.h>

void *calloc(size_t *nmemb*, **size_t** *size*);
void *malloc(size_t *size*);
void free(void **ptr*);
void *realloc(void **ptr*, **size_t** *size*);

DESCRIPTION

calloc() allocates memory for an array of *nmemb* elements of *size* bytes each and returns a pointer to the allocated memory. The memory is set to zero.

malloc() allocates *size* bytes and returns a pointer to the allocated memory. The memory is not cleared.

free() frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc**(), **calloc**() or **realloc**(). Otherwise, or if **free**(*ptr*) has already been called before, undefined behaviour occurs. If *ptr* is **NULL**, no operation is performed.

realloc() changes the size of the memory block pointed to by *ptr* to *size* bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If *ptr* is **NULL**, the call is equivalent to **malloc(size)**; if size is equal to zero, the call is equivalent to **free**(*ptr*). Unless *ptr* is **NULL**, it must have been returned by an earlier call to **malloc**(), **calloc**() or **realloc**().

RETURN VALUE

For **calloc**() and **malloc**(), the value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable, or **NULL** if the request fails.

free() returns no value.

realloc() returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from *ptr*, or **NULL** if the request fails. If *size* was equal to 0, either NULL or a pointer suitable to be passed to *free*() is returned. If **realloc**() fails the original block is left untouched - it is not freed or moved.

CONFORMING TO

ANSI-C

SEE ALSO

brk(2), **posix_memalign**(3)

NAME

sigaction – POSIX signal handling functions.

SYNOPSIS

#include <signal.h>

int sigaction(int *signum*, **const struct sigaction** **act*, **struct sigaction** **oldact*);

DESCRIPTION

The **sigaction** system call is used to change the action taken by a process on receipt of a specific signal.

signum specifies the signal and can be any valid signal except **SIGKILL** and **SIGSTOP**.

If *act* is non–null, the new action for signal *signum* is installed from *act*. If *oldact* is non–null, the previous action is saved in *oldact*.

The **sigaction** structure is defined as something like

```
struct sigaction {
    void     (*sa_handler)(int signal_number);
    sigset_t sa_mask;
    int      sa_flags;
}
```

sa_handler specifies the action to be associated with *signum* and may be **SIG_DFL** for the default action, **SIG_IGN** to ignore this signal, or a pointer to a signal handling function.

sa_mask gives a mask of signals which should be blocked during execution of the signal handler. In addition, the signal which triggered the handler will be blocked, unless the **SA_NODEFER** or **SA_NOMASK** flags are used.

sa_flags specifies a set of flags which modify the behaviour of the signal handling process. It is formed by the bitwise OR of zero or more of the following:

SA_NOCLDSTOP
: If *signum* is **SIGCHLD**, do not receive notification when child processes stop (i.e., when child processes receive one of **SIGSTOP**, **SIGTSTP**, **SIGTTIN** or **SIGTTOU**).

SA_RESTART
: Provide behaviour compatible with BSD signal semantics by making certain system calls restartable across signals. Without **SA_RESTART** the system calls return an error and set errno to EINTR when interrupted by a signal.

RETURN VALUES

sigaction() returns 0 on success; on error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
: An invalid signal was specified. This will also be generated if an attempt is made to change the action for **SIGKILL** or **SIGSTOP**, which cannot be caught.

SEE ALSO

kill(1), **kill**(2), **killpg**(2), **pause**(2), **sigsetops**(3),

NAME

sigprocmask – change and/or examine caller's signal mask
sigsuspend – install a signal mask and suspend caller until signal

SYNOPSIS

#include <signal.h>

int sigprocmask(int *how*, **const sigset_t** **set*, **sigset_t** **oset*);

int sigsuspend(const sigset_t **set*);

DESCRIPTION sigprocmask

The **sigprocmask( )** function is used to examine and/or change the caller's signal mask. If the value is **SIG_BLOCK**, the set pointed to by the argument *set* is added to the current signal mask. If the value is **SIG_UNBLOCK**, the set pointed by the argument *set* is removed from the current signal mask. If the value is **SIG_SETMASK**, the current signal mask is replaced by the set pointed to by the argument *set*. If the argument *oset* is not NULL, the previous mask is stored in the space pointed to by *oset*. If the value of the argument *set* is NULL, the value *how* is not significant and the caller's signal mask is unchanged; thus, the call can be used to inquire about currently blocked signals.

If there are any pending unblocked signals after the call to **sigprocmask( )**, at least one of those signals will be delivered before the call to **sigprocmask( )** returns.

It is not possible to block those signals that cannot be ignored this restriction is silently imposed by the system. See **sigaction**(2).

If **sigprocmask( )** fails, the caller's signal mask is not changed.

RETURN VALUES

On success, **sigprocmask( )** returns **0**. On failure, it returns **–1** and sets **errno** to indicate the error.

ERRORS

sigprocmask( ) fails if any of the following is true:

EFAULT *set* or *oset* points to an illegal address.

EINVAL The value of the *how* argument is not equal to one of the defined values.

DESCRIPTION sigsuspend

sigsuspend( ) replaces the caller's signal mask with the set of signals pointed to by the argument *set* and then suspends the caller until delivery of a signal whose action is either to execute a signal catching function or to terminate the process.

If the action is to terminate the process, **sigsuspend( )** does not return. If the action is to execute a signal catching function, **sigsuspend( )** returns after the signal catching function returns. On return, the signal mask is restored to the set that existed before the call to **sigsuspend( )**.

It is not possible to block those signals that cannot be ignored (see **signal**(5)); this restriction is silently imposed by the system.

RETURN VALUES

Since **sigsuspend( )** suspends process execution indefinitely, there is no successful completion return value. On failure, it returns –1 and sets **errno** to indicate the error.

ERRORS

sigsuspend( ) fails if either of the following is true:

EFAULT *set* points to an illegal address.

EINTR A signal is caught by the calling process and control is returned from the signal catching function.

SEE ALSO

sigaction(2), **sigsetops**(3C),

NAME

sigsetops, sigemptyset, sigfillset, sigaddset, sigdelset, sigismember – manipulate sets of signals

SYNOPSIS

#include <signal.h>

int sigemptyset(sigset_t **set*);

int sigfillset(sigset_t **set*);

int sigaddset(sigset_t **set*, **int** *signo*);

int sigdelset(sigset_t **set*, **int** *signo*);

int sigismember(sigset_t **set*, **int** *signo*);

DESCRIPTION

These functions manipulate *sigset_t* data types, representing the set of signals supported by the implementation.

sigemptyset( ) initializes the set pointed to by *set* to exclude all signals defined by the system.

sigfillset( ) initializes the set pointed to by *set* to include all signals defined by the system.

sigaddset( ) adds the individual signal specified by the value of *signo* to the set pointed to by *set*.

sigdelset( ) deletes the individual signal specified by the value of *signo* from the set pointed to by *set*.

sigismember( ) checks whether the signal specified by the value of *signo* is a member of the set pointed to by *set*.

Any object of type *sigset_t* must be initialized by applying either **sigemptyset( )** or **sigfillset( )** before applying any other operation.

RETURN VALUES

Upon successful completion, the **sigismember( )** function returns a value of one if the specified signal is a member of the specified set, or a value of 0 if it is not. Upon successful completion, the other functions return a value of 0. Otherwise a value of –1 is returned and **errno** is set to indicate the error.

ERRORS

sigaddset( ), **sigdelset( )**, and **sigismember( )** will fail if the following is true:

EINVAL The value of the *signo* argument is not a valid signal number.

sigfillset( ) will fail if the following is true:

EFAULT The *set* argument specifies an invalid address.

SEE ALSO

sigaction(2), **sigpending**(2), **sigprocmask**(2), **sigsuspend**(2), **attributes**(5), **signal**(5)

NAME

waitpid – wait for child process to change state

SYNOPSIS

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

pid_t waitpid(pid_t *pid*, **int** **stat_loc*, **int** *options*);

DESCRIPTION

waitpid( ) suspends the calling process until one of its children changes state; if a child process changed state prior to the call to **waitpid**( ), return is immediate. *pid* specifies a set of child processes for which status is requested.

If *pid* is equal to **(pid_t)−1**, status is requested for any child process.

If *pid* is greater than **(pid_t)0**, it specifies the process ID of the child process for which status is requested.

If *pid* is equal to **(pid_t)0** status is requested for any child process whose process group ID is equal to that of the calling process.

If *pid* is less than **(pid_t)−1**, status is requested for any child process whose process group ID is equal to the absolute value of *pid*.

If **waitpid**( ) returns because the status of a child process is available, then that status may be evaluated with the macros defined by **wstat**(5). If the calling process had specified a non-zero value of *stat_loc*, the status of the child process will be stored in the location pointed to by *stat_loc*.

The *options* argument is constructed from the bitwise inclusive OR of zero or more of the following flags, defined in the header **<sys/wait.h>**:

WCONTINUED The status of any continued child process specified by *pid*, whose status has not been reported since it continued, is also reported to the calling process.

WNOHANG **waitpid**( ) will not suspend execution of the calling process if status is not immediately available for one of the child processes specified by *pid*.

WNOWAIT Keep the process whose status is returned in *stat_loc* in a waitable state. The process may be waited for again with identical results.

If *wstatus* is not NULL, **wait**() and **waitpid**() store status information in the *int* to which it points. This integer can be inspected with the following macros (which take the integer itself as an argument, not a pointer to it, as is done in **wait**() and **waitpid**()!):

WIFEXITED(*wstatus*)
returns true if the child terminated normally, that is, by calling **exit**(3) or **_exit**(2), or by returning from main().

WEXITSTATUS(*wstatus*)
returns the exit status of the child. This consists of the least significant 8 bits of the *status* argument that the child specified in a call to **exit**(3) or **_exit**(2) or as the argument for a return statement in main(). This macro should be employed only if **WIFEXITED** returned true.

WIFSIGNALED(*wstatus*)
returns true if the child process was terminated by a signal.

WTERMSIG(*wstatus*)
returns the number of the signal that caused the child process to terminate. This macro should be employed only if **WIFSIGNALED** returned true.

RETURN VALUES

If **waitpid**( ) returns because the status of a child process is available, this function returns a value equal to the process ID of the child process for which status is reported. If **waitpid**( ) returns due to the delivery of a signal to the calling process, **−1** is returned and **errno** is set to **EINTR**. If this function was invoked with

WNOHANG set in *options*, it has at least one child process specified by *pid* for which status is not available, and status is not available for any process specified by *pid*, **0** is returned. Otherwise, **−1** is returned, and **errno** is set to indicate the error.

ERRORS

waitpid( ) will fail if one or more of the following is true:

ECHILD The process or process group specified by *pid* does not exist or is not a child of the calling process or can never be in the states specified by *options*.

EINTR **waitpid**( ) was interrupted due to the receipt of a signal sent by the calling process.

EINVAL An invalid value was specified for *options*.

SEE ALSO

exec(2), **exit**(2), **fork**(2), **sigaction**(2)