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 VALUES

The **accept( )** function returns **–1** on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

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. 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 VALUES

If the bind is successful, **0** is returned. A return value of **–1** indicates an error, which is further specified in the global **errno**.

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

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

SYNOPSIS

#include <unistd.h>

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

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

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

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

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

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

DESCRIPTION

Each of the functions in the **exec** family overlays a new process image on an old process. The new process image is constructed from an ordinary, executable file. This file is either an executable object file, or a file of data for an interpreter. There can be no return from a successful call to one of these functions because the calling process image is overlaid by the new process image.

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

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

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

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

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

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

The *file* argument points to the new process file. If *file* does not contain a slash character, the path prefix for this file is obtained by a search of the directories passed in the **PATH** environment variable (see **environ**(5)).

File descriptors open in the calling process remain open in the new process.

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

RETURN VALUES

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

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

fopen – open a stream

SYNOPSIS

#include <stdio.h>

FILE *fopen(const char * *filename***, const char *** *mode***);**

DESCRIPTION

The **fopen( )** function opens the file whose pathname is the string pointed to by *filename*, and associates a stream with it.

The argument *mode* points to a string beginning with one of the following sequences:

r or **rb**	open file for reading
w or **wb**	truncate to zero length or create file for writing
a or **ab**	append; open or create file for writing at end-of-file
r+ or **rb+** or **r+b**	open file for update (reading and writing)
w+ or **wb+** or **w+b**	truncate to zero length or create file for update
a+ or **ab+** or **a+b**	append; open or create file for update, writing at end-of-file

The character **b** has no effect, but is allowed for ISO C standard conformance. Opening a file with read mode (**r** as the first character in the *mode* argument) fails if the file does not exist or cannot be read.

When a file is opened with update mode (**+** as the second or third character in the *mode* argument), both input and output may be performed on the associated stream. However, output must not be directly followed by input without an intervening call to **fflush**(3S) or to a file positioning function (**fseek**(3S), **fsetpos**(3S) or **rewind**(3S)), and input must not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file.

When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for the stream are cleared.

If *mode* is **w**, **a**, **w+** or **a+** and the file did not previously exist, upon successful completion, **fopen( )** function will mark for update the **st_atime**, **st_ctime** and **st_mtime** fields of the file and the **st_ctime** and **st_mtime** fields of the parent directory.

If *mode* is **w** or **w+** and the file did previously exist, upon successful completion, **fopen( )** will mark for update the **st_ctime** and **st_mtime** fields of the file. The **fopen( )** function will allocate a file descriptor as **open**(2) does.

The largest value that can be represented correctly in an object of type **off_t** will be established as the offset maximum in the open file description.

RETURN VALUES

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

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

ERRORS

The **fopen( )** function will fail if:

EACCES Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *mode* are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.

EINTR A signal was caught during **fopen( )**.

EISDIR The named file is a directory and *mode* requires write access.

SEE ALSO

fclose(3S), **fdopen**(3S), **fflush**(3S), **freopen**(3S), **fsetpos**(3S), **rewind**(3S),

NAME

ip – Linux IPv4 protocol implementation

SYNOPSIS

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

tcp_socket **= socket(PF_INET, SOCK_STREAM, 0);**
raw_socket **= socket(PF_INET, SOCK_RAW,** *protocol***);**
udp_socket **= socket(PF_INET, SOCK_DGRAM,** *protocol***);**

DESCRIPTION

The programmer's interface is BSD sockets compatible. For more information on sockets, see **socket**(7).

An IP socket is created by calling the **socket**(2) function as **socket(PF_INET, socket_type, protocol)**. Valid socket types are **SOCK_STREAM** to open a **tcp**(7) socket, **SOCK_DGRAM** to open a **udp**(7) socket, or **SOCK_RAW** to open a **raw**(7) socket to access the IP protocol directly. *protocol* is the IP protocol in the IP header to be received or sent. The only valid values for *protocol* are **0** and **IPPROTO_TCP** for TCP sockets and **0** and **IPPROTO_UDP** for UDP sockets.

When a process wants to receive new incoming packets or connections, it should bind a socket to a local interface address using **bind**(2). Only one IP socket may be bound to any given local (address, port) pair. When **INADDR_ANY** is specified in the bind call the socket will be bound to *all* local interfaces. When **listen**(2) or **connect**(2) are called on a unbound socket the socket is automatically bound to a random free port with the local address set to **INADDR_ANY**.

ADDRESS FORMAT

An IP socket address is defined as a combination of an IP interface address and a port number. The basic IP protocol does not supply port numbers, they are implemented by higher level protocols like **tcp**(7).

```
struct sockaddr_in {
    sa_family_t     sin_family;   /* address family: AF_INET */
    u_int16_t       sin_port;     /* port in network byte order */
    struct in_addr  sin_addr;     /* internet address */
};
/* Internet address. */
struct in_addr {
    u_int32_t       s_addr;       /* address in network byte order */
};
```

sin_family is always set to **AF_INET**. This is required; in Linux 2.2 most networking functions return **EINVAL** when this setting is missing. *sin_port* contains the port in network byte order. The port numbers below 1024 are called *reserved ports*. Only processes with effective user id 0 or the **CAP_NET_BIND_SERVICE** capability may **bind**(2) to these sockets.

sin_addr is the IP host address. The *addr* member of **struct in_addr** contains the host interface address in network order. **in_addr** should be only accessed using the **inet_aton**(3), **inet_addr**(3), **inet_makeaddr**(3) library functions or directly with the name resolver (see **gethostbyname**(3)).

Note that the address and the port are always stored in network order. In particular, this means that you need to call **htons**(3) on the number that is assigned to a port. All address/port manipulation functions in the standard library work in network order.

SEE ALSO

sendmsg(2), **recvmsg**(2), **socket**(7), **netlink**(7), **tcp**(7), **udp**(7), **raw**(7), **ipfw**(7)

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);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}
```

On some architectures a union is involved - do not assign to both *sa_handler* and *sa_sigaction*.

The *sa_restorer* element is obsolete and should not be used. POSIX does not specify a *sa_restorer* element.

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.

RETURN VALUES

sigaction returns 0 on success and -1 on 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

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

socket – create an endpoint for communication

SYNOPSIS

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

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

int socket(int *domain***, int** *type***, int** *protocol***);**

DESCRIPTION

socket( ) creates an endpoint for communication and returns a descriptor.

The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file **<sys/socket.h>**. There must be an entry in the **netconfig**(4) file for at least each protocol family and type required. If *protocol* has been specified, but no exact match for the tuplet family, type, protocol is found, then the first entry containing the specified family and type with zero for protocol will be used. The currently understood formats are:

PF_UNIX UNIX system internal protocols

PF_INET ARPA Internet protocols

The socket has the indicated *type*, which specifies the communication semantics. Currently defined types are:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM

A **SOCK_STREAM** type provides sequenced, reliable, two-way connection-based byte streams. An out-of-band data transmission mechanism may be supported. A **SOCK_DGRAM** socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). A **SOCK_SEQPACKET** socket may provide a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer may be required to read an entire packet with each read system call. This facility is protocol specific, and presently not implemented for any protocol family. **SOCK_RAW** sockets provide access to internal network interfaces. The types **SOCK_RAW**, which is available only to the super-user, and **SOCK_RDM**, for which no implementation currently exists, are not described here.

protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, multiple protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the "communication domain" in which communication is to take place. If a protocol is specified by the caller, then it will be packaged into a socket level option request and sent to the underlying protocol layers.

Sockets of type **SOCK_STREAM** are full-duplex byte streams, similar to pipes. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with a **connect**(3N) call. Once connected, data may be transferred using **read**(2) and **write**(2) calls or some variant of the **send**(3N) and **recv**(3N) calls. When a session has been completed, a **close**(2) may be performed. Out-of-band data may also be transmitted as described on the **send**(3N) manual page and received as described on the **recv**(3N) manual page.

The communications protocols used to implement a **SOCK_STREAM** insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with –1 returns and with **ETIMEDOUT** as the specific code in the global variable **errno**. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other

activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (for instance 5 minutes). A **SIGPIPE** signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_SEQPACKET sockets employ the same system calls as **SOCK_STREAM** sockets. The only difference is that **read**(2) calls will return only the amount of data requested, and any remaining in the arriving packet will be discarded.

SOCK_DGRAM and **SOCK_RAW** sockets allow datagrams to be sent to correspondents named in **sendto**(3N) calls. Datagrams are generally received with **recvfrom**(3N), which returns the next datagram with its return address.

An **fcntl**(2) call can be used to specify a process group to receive a **SIGURG** signal when the out-of-band data arrives. It may also enable non-blocking I/O and asynchronous notification of I/O events with **SIGIO** signals.

The operation of sockets is controlled by socket level *options*. These options are defined in the file **<sys/socket.h>**. **setsockopt**(3N) and **getsockopt**(3N) are used to set and get options, respectively.

RETURN VALUES

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

ERRORS

The **socket( )** call fails if:

EACCES	Permission to create a socket of the specified type and/or protocol is denied.
EMFILE	The per-process descriptor table is full.
ENOMEM	Insufficient user memory is available.
ENOSR	There were insufficient STREAMS resources available to complete the operation.
EPROTONOSUPPORT	The protocol type or the specified protocol is not supported within this domain.

SEE ALSO

close(2), **fcntl**(2), **ioctl**(2), **read**(2), **write**(2), **accept**(3N), **bind**(3N), **connect**(3N), **getsockname**(3N), **getsockopt**(3N), **listen**(3N), **recv**(3N), **setsockopt**(3N), **send**(3N), **shutdown**(3N), **socketpair**(3N), **attributes**(5), **in**(5), **socket**(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.

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), **wstat**(5)

NAME

wstat – wait status

SYNOPSIS

#include <sys/wait.h>

DESCRIPTION

When a process waits for status from its children via either the **wait** or **waitpid** function, the status returned may be evaluated with the following macros, defined in **<sys/wait.h>**. These macros evaluate to integral expressions. The *stat* argument to these macros is the integer value returned from **wait** or **waitpid**.

WIFEXITED(*stat***)** Evaluates to a non-zero value if status was returned for a child process that terminated normally.

WEXITSTATUS(*stat***)** If the value of **WIFEXITED(***stat***)** is non-zero, this macro evaluates to the exit code that the child process passed to **_exit( )** (see **exit**(2)) or **exit**(3C), or the value that the child process returned from **main**.

WIFSIGNALED(*stat***)** Evaluates to a non-zero value if status was returned for a child process that terminated due to the receipt of a signal.

WTERMSIG(*stat***)** If the value of **WIFSIGNALED(***stat***)** is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFSTOPPED(*stat***)** Evaluates to a non-zero value if status was returned for a child process that is currently stopped.

WSTOPSIG(*stat***)** If the value of **WIFSTOPPED(***stat***)** is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

WIFCONTINUED(*stat***)** Evaluates to a non-zero value if status was returned for a child process that has continued.

WCOREDUMP(*stat***)** If the value of **WIFSIGNALED** (*stat*) is non-zero, this macro evaluates to a non-zero value if a core image of the terminated child was created.

SEE ALSO

exit(2), **wait**(2), **waitpid**(2), **exit**(3C)