NAME

connect – initiate a connection on a socket

SYNOPSIS

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

int connect(int *sockfd*, **const struct sockaddr** **addr*,
socklen_t *addrlen*);

DESCRIPTION

The **connect**() system call connects the socket referred to by the file descriptor *sockfd* to the address specified by *addr*. The *addrlen* argument specifies the size of *addr*. The format of the address in *addr* is determined by the address space of the socket *sockfd*; see **socket**(2) for further details.

If the socket *sockfd* is of type **SOCK_DGRAM**, then *addr* is the address to which datagrams are sent by default, and the only address from which datagrams are received. If the socket is of type **SOCK_STREAM** or **SOCK_SEQPACKET**, this call attempts to make a connection to the socket that is bound to the address specified by *addr*.

Some protocol sockets (e.g., UNIX domain stream sockets) may successfully **connect**() only once.

Some protocol sockets (e.g., datagram sockets in the UNIX and Internet domains) may use **connect**() multiple times to change their association.

Some protocol sockets (e.g., TCP sockets as well as datagram sockets in the UNIX and Internet domains) may dissolve the association by connecting to an address with the *sa_family* member of *sockaddr* set to **AF_UNSPEC**; thereafter, the socket can be connected to another address. (**AF_UNSPEC** is supported on Linux since kernel 2.2.)

RETURN VALUE

If the connection or binding succeeds, zero is returned. On error, −1 is returned, and *errno* is set appropriately.

ERRORS

The following are some general socket errors only. There may be other domain-specific error codes.

EACCES
For UNIX domain sockets, which are identified by pathname: Write permission is denied on the socket file, or search permission is denied for one of the directories in the path prefix. (See also **path_resolution**(7).)

EBADF
sockfd is not a valid open file descriptor.

ECONNREFUSED
A **connect**() on a stream socket found no one listening on the remote address.

EINTR
The system call was interrupted by a signal that was caught; see **signal**(7).

ENETUNREACH
Network is unreachable.

NOTES

If **connect**() fails, consider the state of the socket as unspecified. Portable applications should close the socket and create a new one for reconnecting.

NAME

getaddrinfo, freeaddrinfo, gai_strerror – network address and service translation

SYNOPSIS

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

int getaddrinfo(const char **node*, **const char** **service*,
const struct addrinfo **hints*,
struct addrinfo ***res*);

void freeaddrinfo(struct addrinfo **res*);

const char ***gai_strerror(int** *errcode*);

DESCRIPTION

Given *node* and *service*, which identify an Internet host and a service, **getaddrinfo** () returns one or more *addrinfo* structures, each of which contains an Internet address that can be specified in a call to **bind (2)** or **connect (2)**.

The *addrinfo* structure used by **getaddrinfo** () contains the following fields:

```
struct addrinfo {
    int              ai_flags;
    int              ai_family;
    int              ai_socktype;
    int              ai_protocol;
    size_t           ai_addrlen;
    struct sockaddr *ai_addr;
    char            *ai_canonname;
    struct addrinfo *ai_next;
};
```

The *hints* argument points to an *addrinfo* structure that specifies criteria for selecting the socket address structures returned in the list pointed to by *res*.

The **freeaddrinfo** () function frees the memory that was allocated for the dynamically allocated linked list res .

RETURN VALUE

getaddrinfo () returns 0 if it succeeds, or one of the following nonzero error codes:

EAI_ADDRFAMILY
The specified network host does not have any network addresses in the requested address family.

EAI_NODATA
The specified network host exists, but does not have any network addresses defined.

EAI_SYSTEM
Other system error, check *errno* for details.

The **gai_strerror**() function translates these error codes to a human readable string, suitable for error reporting.

SEE ALSO

gethostbyname(3), **getnameinfo**(3), **inet**(3), **hostname**(7), **ip**(7)

NAME

kill − send signal to a process

SYNOPSIS

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

int kill(pid_t *pid***, int** *sig***);**

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):

kill(): _POSIX_C_SOURCE

DESCRIPTION

The **kill**() system call can be used to send any signal to any process group or process.

If *pid* is positive, then signal *sig* is sent to the process with the ID specified by *pid*.

If *pid* equals 0, then *sig* is sent to every process in the process group of the calling process.

If *pid* equals −1, then *sig* is sent to every process for which the calling process has permission to send signals, except for process 1 (*init*), but see below.

If *pid* is less than −1, then *sig* is sent to every process in the process group whose ID is −*pid*.

If *sig* is 0, then no signal is sent, but existence and permission checks are still performed; this can be used to check for the existence of a process ID or process group ID that the caller is permitted to signal.

For a process to have permission to send a signal, it must either be privileged (under Linux: have the **CAP_KILL** capability in the user namespace of the target process), or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the target process. In the case of **SIGCONT**, it suffices when the sending and receiving processes belong to the same session. (Historically, the rules were different; see NOTES.)

RETURN VALUE

On success (at least one signal was sent), zero is returned. On error, −1 is returned, and *errno* is set appropriately.

ERRORS

EINVAL
An invalid signal was specified.

EPERM
The calling process does not have permission to send the signal to any of the target processes.

ESRCH
The target process or process group does not exist. Note that an existing process might be a zombie, a process that has terminated execution, but has not yet been **wait**(2)ed for.

NOTES

The only signals that can be sent to process ID 1, the *init* process, are those for which *init* has explicitly installed signal handlers. This is done to assure the system is not brought down accidentally.

POSIX.1 requires that *kill(−1,sig)* send *sig* to all processes that the calling process may send signals to, except possibly for some implementation-defined system processes. Linux allows a process to signal itself, but on Linux the call *kill(−1,sig)* does not signal the calling process.

POSIX.1 requires that if a process sends a signal to itself, and the sending thread does not have the signal blocked, and no other thread has it unblocked or is waiting for it in **sigwait**(3), at least one unblocked signal must be delivered to the sending thread before the **kill**() returns.

NAME

pthread_create − create a new thread / pthread_exit − terminate the calling thread

SYNOPSIS

#include <pthread.h>

int pthread_create(pthread_t * *thread***, pthread_attr_t *** *attr***, void * (***start_routine***)(void *), void *** *arg***);**

void pthread_exit(void * *retval***);**

DESCRIPTION

pthread_create creates a new thread of control that executes concurrently with the calling thread. The new thread applies the function *start_routine* passing it *arg* as first argument. The new thread terminates either explicitly, by calling **pthread_exit**(3), or implicitly, by returning from the *start_routine* function. The latter case is equivalent to calling **pthread_exit**(3) with the result returned by *start_routine* as exit code.

The *attr* argument specifies thread attributes to be applied to the new thread. See **pthread_attr_init**(3) for a complete list of thread attributes. The *attr* argument can also be **NULL**, in which case default attributes are used: the created thread is joinable (not detached) and has default (non real-time) scheduling policy.

pthread_exit terminates the execution of the calling thread. All cleanup handlers that have been set for the calling thread with **pthread_cleanup_push**(3) are executed in reverse order (the most recently pushed handler is executed first). Finalization functions for thread-specific data are then called for all keys that have non- **NULL** values associated with them in the calling thread (see **pthread_key_create**(3)). Finally, execution of the calling thread is stopped.

The *retval* argument is the return value of the thread. It can be consulted from another thread using **pthread_join**(3).

RETURN VALUE

On success, the identifier of the newly created thread is stored in the location pointed by the *thread* argument, and a 0 is returned. On error, a non-zero error code is returned.

The **pthread_exit** function never returns.

ERRORS

EAGAIN
not enough system resources to create a process for the new thread.

EAGAIN
more than **PTHREAD_THREADS_MAX** threads are already active.

AUTHOR

Xavier Leroy <Xavier.Leroy@inria.fr>

SEE ALSO

pthread_join(3), **pthread_detach**(3), **pthread_attr_init**(3).

NAME

pthread_join − join with a terminated thread

SYNOPSIS

#include <pthread.h>

int pthread_join(pthread_t *thread*, **void ****retval**);**

Compile and link with −*pthread*.

DESCRIPTION

The **pthread_join**() function waits for the thread specified by *thread* to terminate. If that thread has already terminated, then **pthread_join**() returns immediately. The thread specified by *thread* must be joinable.

If *retval* is not NULL, then **pthread_join**() copies the exit status of the target thread (i.e., the value that the target thread supplied to **pthread_exit**(3)) into the location pointed to by *retval*. If the target thread was canceled, then **PTHREAD_CANCELED** is placed in the location pointed to by *retval*.

If multiple threads simultaneously try to join with the same thread, the results are undefined. If the thread calling **pthread_join**() is canceled, then the target thread will remain joinable (i.e., it will not be detached).

RETURN VALUE

On success, **pthread_join**() returns 0; on error, it returns an error number.

ERRORS

EDEADLK
A deadlock was detected (e.g., two threads tried to join with each other); or *thread* specifies the calling thread.

EINVAL
thread is not a joinable thread.

EINVAL
Another thread is already waiting to join with this thread.

ESRCH
No thread with the ID *thread* could be found.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
pthread_join()	Thread safety	MT-Safe

NOTES

After a successful call to **pthread_join**(), the caller is guaranteed that the target thread has terminated. The caller may then choose to do any clean-up that is required after termination of the thread (e.g., freeing memory or other resources that were allocated to the target thread).

Joining with a thread that has previously been joined results in undefined behavior.

Failure to join with a thread that is joinable (i.e., one that is not detached), produces a "zombie thread". Avoid doing this, since each zombie thread consumes some system resources, and when enough zombie threads have accumulated, it will no longer be possible to create new threads (or processes).

There is no pthreads analog of *waitpid(-1, &status, 0)*, that is, "join with any terminated thread". If you believe you need this functionality, you probably need to rethink your application design.

All of the threads in a process are peers: any thread can join with any other thread in the process.

NAME

pthread_sigmask − examine and change mask of blocked signals

SYNOPSIS

#include <signal.h>

int pthread_sigmask(int *how*, **const sigset_t ***set, **sigset_t ***oldset**);**

Compile and link with −*pthread*.

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):

pthread_sigmask():
_POSIX_C_SOURCE >= 199506L || _XOPEN_SOURCE >= 500

DESCRIPTION

The **pthread_sigmask**() function is just like **sigprocmask**(2), with the difference that its use in multithreaded programs is explicitly specified by POSIX.1. Other differences are noted in this page.

For a description of the arguments and operation of this function, see **sigprocmask**(2).

RETURN VALUE

On success, **pthread_sigmask**() returns 0; on error, it returns an error number.

ERRORS

See **sigprocmask**(2).

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
pthread_sigmask()	Thread safety	MT-Safe

NOTES

A new thread inherits a copy of its creator's signal mask.

The glibc **pthread_sigmask**() function silently ignores attempts to block the two real-time signals that are used internally by the NPTL threading implementation. See **nptl**(7) for details.

NAME

fputc, fputs, putc, putchar, puts – output of characters and strings

SYNOPSIS

```
#include <stdio.h>

int fputc(int c, FILE *stream);

int fputs(const char *s, FILE *stream);

int putc(int c, FILE *stream);

int putchar(int c);

int puts(const char *s);
```

DESCRIPTION

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*).

puts() writes the string *s* and a trailing newline to *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.

For nonlocking counterparts, see **unlocked_stdio**(3).

RETURN VALUE

fputc(), **putc**(), and **putchar**() return the character written as an *unsigned char* cast to an *int* or **EOF** on error.

puts() and **fputs**() return a nonnegative number on success, or **EOF** on error.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
fputc(), **fputs**(), **putc**(), **putchar**(), **puts**()	Thread safety	MT-Safe

NAME

scandir, alphasort, versionsort – scan a directory for matching entries

SYNOPSIS

```
#include <dirent.h>

int scandir(const char *dirp, struct dirent ***namelist,
            int (*filter)(const struct dirent *),
            int (*compar)(const struct dirent **, const struct dirent **));

int alphasort(const struct dirent **a, const struct dirent **b);

int versionsort(const struct dirent **a, const struct dirent **b);
```

DESCRIPTION

The **scandir**() function scans the directory *dirp*, calling *filter*() on each directory entry. Entries for which *filter*() returns nonzero are stored in strings allocated via **malloc**(3), sorted using **qsort**(3) with the comparison function *compar*(), and collected in array *namelist* which is allocated via **malloc**(3). If *filter* is NULL, all entries are selected.

The **alphasort**() and **versionsort**() functions can be used as the comparison function *compar*(). The former sorts directory entries using **strcoll**(3), the latter using **strverscmp**(3) on the strings *(*a)->d_name* and *(*b)->d_name*.

RETURN VALUE

The **scandir**() function returns the number of directory entries selected. On error, −1 is returned, with *errno* set to indicate the cause of the error.

The **alphasort**() and **versionsort**() functions return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

ERRORS

ENOENT
: The path in *dirp* does not exist.

ENOMEM
: Insufficient memory to complete the operation.

ENOTDIR
: The path in *dirp* is not a directory.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
scandir()	Thread safety	MT-Safe
alphasort(), **versionsort**()	Thread safety	MT-Safe locale

NAME

socket – create an endpoint for communication

SYNOPSIS

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

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

DESCRIPTION

socket() creates an endpoint for communication and returns a file descriptor that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

The *domain* argument specifies a communication domain; this selects the protocol family which will be used for communication. These families are defined in <*sys/socket.h*>. The formats currently understood by the Linux kernel include:

Name	Purpose	Man page
AF_UNIX	Local communication	**unix**(7)
AF_LOCAL	Synonym for **AF_UNIX**	
AF_INET	IPv4 Internet protocols	**ip**(7)
AF_INET6	IPv6 Internet protocols	**ipv6**(7)
AF_XDP	XDP (express data path) interface	

Further details of the above address families, as well as information on several other address families, can be found in **address_families**(7).

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

SOCK_STREAM
Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported.

SOCK_DGRAM Supports datagrams (connectionless, unreliable messages of a fixed maximum length).

Some socket types may not be implemented by all protocol families.

Since Linux 2.6.27, the *type* argument serves a second purpose: in addition to specifying a socket type, it may include the bitwise OR of any of the following values, to modify the behavior of **socket**():

SOCK_NONBLOCK
Set the **O_NONBLOCK** file status flag on the open file description (see **open**(2)) referred to by the new file descriptor. Using this flag saves extra calls to **fcntl**(2) to achieve the same result.

SOCK_CLOEXEC
Set the close-on-exec (**FD_CLOEXEC**) flag on the new file descriptor. See the description of the **O_CLOEXEC** flag in **open**(2) for reasons why this may be useful.

The *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, in which case *protocol* can be specified as 0. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is specific to the "communication domain" in which communication is to take place; see **protocols**(5). See **getprotoent**(3) on how to map protocol name strings to protocol numbers.

Sockets of type **SOCK_STREAM** are full-duplex byte streams. They do not preserve record boundaries. 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**(2) call. Once connected, data may be transferred using **read**(2) and **write**(2) calls or some variant of the **send**(2) and **recv**(2) calls. When a session has been completed a **close**(2) may be performed. Out-of-band data may also be transmitted as described in **send**(2) and received as described in **recv**(2).

The communications protocols which implement a **SOCK_STREAM** ensure 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 to be dead. When **SO_KEEPALIVE** is enabled on the socket the protocol checks in a protocol-specific manner if the other end is still alive. A **SIGPIPE** signal is raised if a process sends or receives 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 data remaining in the arriving packet will be discarded. Also all message boundaries in incoming datagrams are preserved.

SOCK_DGRAM and **SOCK_RAW** sockets allow sending of datagrams to correspondents named in **sendto**(2) calls. Datagrams are generally received with **recvfrom**(2), which returns the next datagram along with the address of its sender.

SOCK_PACKET is an obsolete socket type to receive raw packets directly from the device driver. Use **packet**(7) instead.

An **fcntl**(2) **F_SETOWN** operation can be used to specify a process or process group to receive a **SIGURG** signal when the out-of-band data arrives or **SIGPIPE** signal when a **SOCK_STREAM** connection breaks unexpectedly. This operation may also be used to set the process or process group that receives the I/O and asynchronous notification of I/O events via **SIGIO**. Using **F_SETOWN** is equivalent to an **ioctl**(2) call with the **FIOSETOWN** or **SIOCSPGRP** argument.

When the network signals an error condition to the protocol module (e.g., using an ICMP message for IP) the pending error flag is set for the socket. The next operation on this socket will return the error code of the pending error. For some protocols it is possible to enable a per-socket error queue to retrieve detailed information about the error; see **IP_RECVERR** in **ip**(7).

The operation of sockets is controlled by socket level *options*. These options are defined in <*sys/socket.h*>. The functions **setsockopt**(2) and **getsockopt**(2) are used to set and get options.

RETURN VALUE

On success, a file descriptor for the new socket is returned. On error, −1 is returned, and *errno* is set appropriately.

ERRORS

EACCES
Permission to create a socket of the specified type and/or protocol is denied.

EAFNOSUPPORT
The implementation does not support the specified address family.

EINVAL
Unknown protocol, or protocol family not available.

EINVAL
Invalid flags in *type*.

EMFILE
The per-process limit on the number of open file descriptors has been reached.

ENFILE
The system-wide limit on the total number of open files has been reached.

ENOBUFS or **ENOMEM**
Insufficient memory is available. The socket cannot be created until sufficient resources are freed.

EPROTONOSUPPORT
The protocol type or the specified protocol is not supported within this domain.

Other errors may be generated by the underlying protocol modules.

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)