NAME
connect – initiate a connection on a socket

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

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

DESCRIPTION
The file descriptor *sockfd* must refer to a socket. If the socket is of type **SOCK_DGRAM** then the *serv_addr* address 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 another socket. The other socket is specified by *serv_addr*, which is an address (of length *addrlen*) in the communications space of the socket. Each communications space interprets the *serv_addr* parameter in its own way.

Generally, connection-based protocol sockets may successfully **connect** only once; connectionless protocol sockets may use **connect** multiple times to change their association. Connectionless sockets may dissolve the association by connecting to an address with the *sa_family* member of **sockaddr** set to **AF_UNSPEC**.

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 general socket errors only. There may be other domain-specific error codes.

EBADF
The file descriptor is not a valid index in the descriptor table.

EFAULT
The socket structure address is outside the user's address space.

ENOTSOCK
The file descriptor is not associated with a socket.

EISCONN
The socket is already connected.

ECONNREFUSED
No one listening on the remote address.

ENETUNREACH
Network is unreachable.

EADDRINUSE
Local address is already in use.

EAFNOSUPPORT
The passed address didn't have the correct address family in its *sa_family* field.

EACCES, EPERM
The user tried to connect to a broadcast address without having the socket broadcast flag enabled or the connection request failed because of a local firewall rule.

SEE ALSO
accept(2), **bind**(2), **listen**(2), **socket**(2), **getsockname**(2)

NAME
opendir – open a directory / readdir – read a directory

SYNOPSIS
#include <sys/types.h>

#include <dirent.h>

DIR *opendir(const char **name***);**

struct dirent *readdir(DIR **dir***);**
int readdir_r(DIR **dirp***, struct dirent** **entry***, struct dirent** ***result***);**

DESCRIPTION opendir
The **opendir()** function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

RETURN VALUE
The **opendir()** function returns a pointer to the directory stream or NULL if an error occurred.

DESCRIPTION readdir
The **readdir()** function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by *dir*. It returns NULL on reaching the end-of-file or if an error occurred.

DESCRIPTION readdir_r
The **readdir_r()** function initializes the structure referenced by *entry* and stores a pointer to this structure in *result*. On successful return, the pointer returned at **result* will have the same value as the argument *entry*. Upon reaching the end of the directory stream, this pointer will have the value NULL.

The data returned by **readdir()** is overwritten by subsequent calls to **readdir()** for the **same** directory stream.

The *dirent* structure is defined as follows:

```
struct dirent {
    long        d_ino;              /* inode number */
    off_t       d_off;             /* offset to the next dirent */
    unsigned short d_reclen;       /* length of this record */
    unsigned char  d_type;    /* type of file */
    char        d_name[256];  /* filename */
};
```

RETURN VALUE
The **readdir()** function returns a pointer to a dirent structure, or NULL if an error occurs or end-of-file is reached.

readdir_r() returns 0 if successful or an error number to indicate failure.

ERRORS
EACCES
Permission denied.

ENOENT
Directory does not exist, or *name* is an empty string.

ENOTDIR
name is not a directory.

NAME

fopen, fdopen – stream open functions

SYNOPSIS

#include <stdio.h>

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

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.

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.

ERRORS

EINVAL

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

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).

SEE ALSO

open(2), **fclose**(3), **fileno**(3)

NAME

gets, fgets – get a string from a stream
fputs, puts – output of strings

SYNOPSIS

#include <stdio.h>

char *gets(char *s**);**

char *fgets(char *s**, int** n**, FILE ***stream**);**

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

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

DESCRIPTION gets/fgets

The **gets( )** function reads characters from the standard input stream (see **intro**(3)), **stdin**, into the array pointed to by *s*, until a newline character is read or an end-of-file condition is encountered. The newline character is discarded and the string is terminated with a null character.

The **fgets( )** function reads characters from the *stream* into the array pointed to by *s*, until $n-1$ characters are read, or a newline character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

When using **gets( )**, if the length of an input line exceeds the size of *s*, indeterminate behavior may result. For this reason, it is strongly recommended that **gets( )** be avoided in favor of **fgets( )**.

RETURN VALUES

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a null pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a null pointer is returned and the error indicator for the stream is set. If end-of-file is encountered, the **EOF** indicator for the stream is set. Otherwise *s* is returned.

ERRORS

The **gets( )** and **fgets( )** functions will fail if data needs to be read and:

EOVERFLOW   The file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the corresponding *stream*.

DESCRIPTION puts/fputs

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

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.

RETURN VALUE

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

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
        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

pthread_cond_init,   pthread_cond_destroy,   pthread_cond_signal,   pthread_cond_broadcast,
pthread_cond_wait, pthread_cond_timedwait – operations on conditions

SYNOPSIS

#include <pthread.h>

pthread_cond_t *cond* = **PTHREAD_COND_INITIALIZER;**

int pthread_cond_init(pthread_cond_t **cond***, pthread_condattr_t ****cond_attr***);**

int pthread_cond_signal(pthread_cond_t **cond***);**

int pthread_cond_broadcast(pthread_cond_t **cond***);**

int pthread_cond_wait(pthread_cond_t **cond***, pthread_mutex_t ****mutex***);**

int pthread_cond_timedwait(pthread_cond_t **cond***, pthread_mutex_t ****mutex***, const struct timespec
****abstime***);**

int pthread_cond_destroy(pthread_cond_t **cond***);**

DESCRIPTION

A condition (short for "condition variable") is a synchronization device that allows threads to suspend
execution and relinquish the processors until some predicate on shared data is satisfied. The basic opera-
tions on conditions are: signal the condition (when the predicate becomes true), and wait for the condition,
suspending the thread execution until another thread signals the condition.

A condition variable must always be associated with a mutex, to avoid the race condition where a thread
prepares to wait on a condition variable and another thread signals the condition just before the first thread
actually waits on it.

pthread_cond_init initializes the condition variable *cond*, using the condition attributes specified in
cond_attr, or default attributes if *cond_attr* is **NULL**. The LinuxThreads implementation supports no
attributes for conditions, hence the *cond_attr* parameter is actually ignored.

Variables of type **pthread_cond_t** can also be initialized statically, using the constant
PTHREAD_COND_INITIALIZER.

pthread_cond_signal restarts one of the threads that are waiting on the condition variable *cond*. If no
threads are waiting on *cond*, nothing happens. If several threads are waiting on *cond*, exactly one is
restarted, but it is not specified which.

pthread_cond_broadcast restarts all the threads that are waiting on the condition variable *cond*. Nothing
happens if no threads are waiting on *cond*.

pthread_cond_wait atomically unlocks the *mutex* (as per **pthread_unlock_mutex**) and waits for the con-
dition variable *cond* to be signaled. The thread execution is suspended and does not consume any CPU time
until the condition variable is signaled. The *mutex* must be locked by the calling thread on entrance to
pthread_cond_wait. Before returning to the calling thread, **pthread_cond_wait** re-acquires *mutex* (as per
pthread_lock_mutex).

Unlocking the mutex and suspending on the condition variable is done atomically. Thus, if all threads
always acquire the mutex before signaling the condition, this guarantees that the condition cannot be

signaled (and thus ignored) between the time a thread locks the mutex and the time it waits on the condition
variable.

pthread_cond_timedwait atomically unlocks *mutex* and waits on *cond*, as **pthread_cond_wait** does, but it
also bounds the duration of the wait. If *cond* has not been signaled within the amount of time specified by
abstime, the mutex *mutex* is re-acquired and **pthread_cond_timedwait** returns the error **ETIMEDOUT**.
The *abstime* parameter specifies an absolute time, with the same origin as **time**(2) and **gettimeofday**(2): an
abstime of 0 corresponds to 00:00:00 GMT, January 1, 1970.

pthread_cond_destroy destroys a condition variable, freeing the resources it might hold. No threads must
be waiting on the condition variable on entrance to **pthread_cond_destroy**. In the LinuxThreads imple-
mentation, no resources are associated with condition variables, thus **pthread_cond_destroy** actually does
nothing except checking that the condition has no waiting threads.

CANCELLATION

pthread_cond_wait and **pthread_cond_timedwait** are cancellation points. If a thread is cancelled while
suspended in one of these functions, the thread immediately resumes execution, then locks again the *mutex*
argument to **pthread_cond_wait** and **pthread_cond_timedwait**, and finally executes the cancellation.
Consequently, cleanup handlers are assured that *mutex* is locked when they are called.

ASYNC-SIGNAL SAFETY

The condition functions are not async-signal safe, and should not be called from a signal handler. In partic-
ular, calling **pthread_cond_signal** or **pthread_cond_broadcast** from a signal handler may deadlock the
calling thread.

RETURN VALUE

All condition variable functions return 0 on success and a non-zero error code on error.

ERRORS

pthread_cond_init, **pthread_cond_signal**, **pthread_cond_broadcast**, and **pthread_cond_wait** never
return an error code.

The **pthread_cond_timedwait** function returns the following error codes on error:

ETIMEDOUT
the condition variable was not signaled until the timeout specified by *abstime*

EINTR
pthread_cond_timedwait was interrupted by a signal

The **pthread_cond_destroy** function returns the following error code on error:

EBUSY
some threads are currently waiting on *cond*.

AUTHOR

Xavier Leroy <Xavier.Leroy@inria.fr>

SEE ALSO

pthread_condattr_init(3),   **pthread_mutex_lock**(3),   **pthread_mutex_unlock**(3),   **gettimeofday**(2),
nanosleep(2).

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_detach – put a running thread in the detached state

SYNOPSIS

    #include <pthread.h>

    int pthread_detach(pthread_t th);

DESCRIPTION

    pthread_detach put the thread *th* in the detached state. This guarantees that the memory resources consumed by *th* will be freed immediately when *th* terminates. However, this prevents other threads from synchronizing on the termination of *th* using **pthread_join**.

    A thread can be created initially in the detached state, using the **detachstate** attribute to **pthread_create**(3). In contrast, **pthread_detach** applies to threads created in the joinable state, and which need to be put in the detached state later.

    After **pthread_detach** completes, subsequent attempts to perform **pthread_join** on *th* will fail. If another thread is already joining the thread *th* at the time **pthread_detach** is called, **pthread_detach** does nothing and leaves *th* in the joinable state.

RETURN VALUE

    On success, 0 is returned. On error, a non-zero error code is returned.

ERRORS

    ESRCH

        No thread could be found corresponding to that specified by *th*

    EINVAL

        the thread *th* is already in the detached state

AUTHOR

    Xavier Leroy <Xavier.Leroy@inria.fr>

SEE ALSO

    pthread_create(3), **pthread_join**(3), **pthread_attr_setdetachstate**(3).

NAME

pthread_mutex_init,   pthread_mutex_lock,   pthread_mutex_trylock,   pthread_mutex_unlock,
pthread_mutex_destroy – operations on mutexes

SYNOPSIS

#include <pthread.h>

pthread_mutex_t *fastmutex* = **PTHREAD_MUTEX_INITIALIZER;**

pthread_mutex_t *recmutex* = **PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;**

pthread_mutex_t *errchkmutex* = **PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;**

int pthread_mutex_init(pthread_mutex_t **mutex***, const pthread_mutexattr_t ****mutexattr***);**

int pthread_mutex_lock(pthread_mutex_t **mutex***);**

int pthread_mutex_trylock(pthread_mutex_t **mutex***);**

int pthread_mutex_unlock(pthread_mutex_t **mutex***);**

int pthread_mutex_destroy(pthread_mutex_t **mutex***);**

DESCRIPTION

A mutex is a MUTual EXclusion device, and is useful for protecting shared data structures from concurrent
modifications, and implementing critical sections and monitors.

A mutex has two possible states: unlocked (not owned by any thread), and locked (owned by one thread). A
mutex can never be owned by two different threads simultaneously. A thread attempting to lock a mutex
that is already locked by another thread is suspended until the owning thread unlocks the mutex first.

pthread_mutex_init initializes the mutex object pointed to by *mutex* according to the mutex attributes
specified in *mutexattr*. If *mutexattr* is **NULL**, default attributes are used instead.

The LinuxThreads implementation supports only one mutex attributes, the *mutex kind*, which is either
"fast", "recursive", or "error checking". The kind of a mutex determines whether it can be locked again
by a thread that already owns it. The default kind is "fast". See **pthread_mutexattr_init**(3) for more
information on mutex attributes.

Variables of type **pthread_mutex_t** can also be initialized statically, using the constants
PTHREAD_MUTEX_INITIALIZER (for fast mutexes), **PTHREAD_RECURSIVE_MUTEX_INI-
TIALIZER_NP** (for recursive mutexes), and **PTHREAD_ERRORCHECK_MUTEX_INITIAL-
IZER_NP** (for error checking mutexes).

pthread_mutex_lock locks the given mutex. If the mutex is currently unlocked, it becomes locked and
owned by the calling thread, and **pthread_mutex_lock** returns immediately. If the mutex is already locked
by another thread, **pthread_mutex_lock** suspends the calling thread until the mutex is unlocked.

If the mutex is already locked by the calling thread, the behavior of **pthread_mutex_lock** depends on the
kind of the mutex. If the mutex is of the "fast" kind, the calling thread is suspended until the mutex is
unlocked, thus effectively causing the calling thread to deadlock. If the mutex is of the "error checking"
kind, **pthread_mutex_lock** returns immediately with the error code **EDEADLK**. If the mutex is of the
"recursive" kind, **pthread_mutex_lock** succeeds and returns immediately, recording the number of times
the calling thread has locked the mutex. An equal number of **pthread_mutex_unlock** operations must be

performed before the mutex returns to the unlocked state.

pthread_mutex_trylock behaves identically to **pthread_mutex_lock**, except that it does not block the
calling thread if the mutex is already locked by another thread (or by the calling thread in the case of a
"fast" mutex). Instead, **pthread_mutex_trylock** returns immediately with the error code **EBUSY**.

pthread_mutex_unlock unlocks the given mutex. The mutex is assumed to be locked and owned by the
calling thread on entrance to **pthread_mutex_unlock**. If the mutex is of the "fast" kind,
pthread_mutex_unlock always returns it to the unlocked state. If it is of the "recursive" kind, it decre-
ments the locking count of the mutex (number of **pthread_mutex_lock** operations performed on it by the
calling thread), and only when this count reaches zero is the mutex actually unlocked.

On "error checking" mutexes, **pthread_mutex_unlock** actually checks at run-time that the mutex is
locked on entrance, and that it was locked by the same thread that is now calling **pthread_mutex_unlock**.
If these conditions are not met, an error code is returned and the mutex remains unchanged. "Fast" and
"recursive" mutexes perform no such checks, thus allowing a locked mutex to be unlocked by a thread
other than its owner. This is non-portable behavior and must not be relied upon.

pthread_mutex_destroy destroys a mutex object, freeing the resources it might hold. The mutex must be
unlocked on entrance. In the LinuxThreads implementation, no resources are associated with mutex objects,
thus **pthread_mutex_destroy** actually does nothing except checking that the mutex is unlocked.

RETURN VALUE

pthread_mutex_init always returns 0. The other mutex functions return 0 on success and a non-zero error
code on error.

ERRORS

The **pthread_mutex_lock** function returns the following error code on error:

EINVAL
the mutex has not been properly initialized.

EDEADLK
the mutex is already locked by the calling thread ("error checking" mutexes only).

The **pthread_mutex_unlock** function returns the following error code on error:

EINVAL
the mutex has not been properly initialized.

EPERM
the calling thread does not own the mutex ("error checking" mutexes only).

The **pthread_mutex_destroy** function returns the following error code on error:

EBUSY
the mutex is currently locked.

AUTHOR

Xavier Leroy <Xavier.Leroy@inria.fr>

SEE ALSO

pthread_mutexattr_init(3), **pthread_mutexattr_setkind_np**(3), **pthread_cancel**(3).

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

unlink – remove directory entry

SYNOPSIS

#include <unistd.h>

int unlink(const char **path***);**

DESCRIPTION

The **unlink( )** function removes a link to a file. It removes the link named by the pathname pointed to by *path* and decrements the link count of the file referenced by the link.

When the file's link count becomes 0 and no process has the file open, the space occupied by the file will be freed and the file will no longer be accessible. If one or more processes have the file open when the last link is removed, the link will be removed before **unlink( )** returns, but the removal of the file contents will be postponed until all references to the file are closed.

RETURN VALUES

Upon successful completion, **0** is returned. Otherwise, **−1** is returned and **errno** is set to indicate the error.

ERRORS

The **unlink( )** function will fail and not unlink the file if:

EACCES	Search permission is denied for a component of the *path* prefix.
EACCES	Write permission is denied on the directory containing the link to be removed.
ENOENT	The named file does not exist or is a null pathname.
ENOTDIR	A component of the *path* prefix is not a directory.
EPERM	The named file is a directory and the effective user of the calling process is not super-user.

SEE ALSO

rm(1), **close**(2), **link**(2), **open**(2), **rmdir**(2),