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

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

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 operations 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 condition 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 implementation, 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 particular, 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

SYNOPSIS

#include <pthread.h>

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

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.

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.

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_exit(3), **pthread_join**(3), **pthread_detach**(3), **pthread_attr_init**(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_INITIALIZER_NP** (for recursive mutexes), and **PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_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 decrements 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

stat, fstat, lstat − get file status

SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char * *file_name*, **struct stat ***buf* **);**
int fstat(int *filedes*, **struct stat ***buf* **);**
int lstat(const char * *file_name*, **struct stat ***buf* **);**

DESCRIPTION

These functions return information about the specified file. You do not need any access rights to the file to get this information but you need search rights to all directories named in the path leading to the file.

stat stats the file pointed to by *file_name* and fills in *buf* .

lstat is identical to **stat**, except in the case of a symbolic link, where the link itself is stat-ed, not the file that it refers to.

fstat is identical to **stat**, only the open file pointed to by *filedes* (as returned by **open**(2)) is stat-ed in place of *file_name*.

They all return a *stat* structure, which contains the following fields:

```
struct stat {
    dev_t       st_dev;     /* device */
    ino_t       st_ino;     /* inode */
    mode_t      st_mode;    /* protection */
    nlink_t     st_nlink;   /* number of hard links */
    uid_t       st_uid;     /* user ID of owner */
    gid_t       st_gid;     /* group ID of owner */
    dev_t       st_rdev;    /* device type (if inode device) */
    off_t       st_size;    /* total size, in bytes */
    blksize_t   st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t    st_blocks;  /* number of blocks allocated */
    time_t      st_atime;   /* time of last access */
    time_t      st_mtime;   /* time of last modification */
    time_t      st_ctime;   /* time of last status change */
};
```

The value *st_size* gives the size of the file (if it is a regular file or a symlink) in bytes. The size of a symlink is the length of the pathname it contains, without trailing NUL.

The value *st_blocks* gives the size of the file in 512-byte blocks. (This may be smaller than *st_size*/512 e.g. when the file has holes.) The value *st_blksize* gives the "preferred" blocksize for efficient file system I/O. (Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.)

Not all of the Linux filesystems implement all of the time fields. Some file system types allow mounting in such a way that file accesses do not cause an update of the *st_atime* field. (See ‘noatime’ in **mount**(8).)

The field *st_atime* is changed by file accesses, e.g. by **execve**(2), **mknod**(2), **pipe**(2), **utime**(2) and **read**(2) (of more than zero bytes). Other routines, like **mmap**(2), may or may not update *st_atime*.

The field *st_mtime* is changed by file modifications, e.g. by **mknod**(2), **truncate**(2), **utime**(2) and **write**(2) (of more than zero bytes). Moreover, *st_mtime* of a directory is changed by the creation or deletion of files in that directory. The *st_mtime* field is *not* changed for changes in owner, group, hard link count, or mode.

The field *st_ctime* is changed by writing or by setting inode information (i.e., owner, group, link count, mode, etc.).

The following POSIX macros are defined to check the file type:

S_ISREG(m)	is it a regular file?
S_ISDIR(m)	directory?
S_ISCHR(m)	character device?
S_ISBLK(m)	block device?
S_ISFIFO(m)	fifo?
S_ISLNK(m)	symbolic link? (Not in POSIX.1-1996.)
S_ISSOCK(m)	socket? (Not in POSIX.1-1996.)

The following flags are defined for the *st_mode* field:

S_IFMT	0170000	bitmask for the file type bitfields
S_IFSOCK	0140000	socket
S_IFLNK	0120000	symbolic link
S_IFREG	0100000	regular file
S_IFBLK	0060000	block device
S_IFDIR	0040000	directory
S_IFCHR	0020000	character device
S_IFIFO	0010000	fifo
S_ISUID	0004000	set UID bit
S_ISGID	0002000	set GID bit (see below)
S_ISVTX	0001000	sticky bit (see below)
S_IRWXU	00700	mask for file owner permissions
S_IRUSR	00400	owner has read permission
S_IWUSR	00200	owner has write permission
S_IXUSR	00100	owner has execute permission
S_IRWXG	00070	mask for group permissions
S_IRGRP	00040	group has read permission
S_IWGRP	00020	group has write permission
S_IXGRP	00010	group has execute permission
S_IRWXO	00007	mask for permissions for others (not in group)
S_IROTH	00004	others have read permission
S_IWOTH	00002	others have write permisson
S_IXOTH	00001	others have execute permission

The set GID bit (S_ISGID) has several special uses: For a directory it indicates that BSD semantics is to be used for that directory: files created there inherit their group ID from the directory, not from the effective group ID of the creating process, and directories created there will also get the S_ISGID bit set. For a file that does not have the group execution bit (S_IXGRP) set, it indicates mandatory file/record locking.

The 'sticky' bit (S_ISVTX) on a directory means that a file in that directory can be renamed or deleted only by the owner of the file, by the owner of the directory, and by a privileged process.

RETURN VALUE

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

SEE ALSO

chmod(2), **chown**(2), **readlink**(2), **utime**(2), **capabilities**(7)

NAME

strerror, strerror_r − return string describing error code

SYNOPSIS

#include <string.h>

char *strerror(int *errnum***);**
int strerror_r(int *errnum***, char ***buf***, size_t** *n***);**

DESCRIPTION

The **strerror()** function returns a string describing the error code passed in the argument *errnum*, possibly using the LC_MESSAGES part of the current locale to select the appropriate language. This string must not be modified by the application, but may be modified by a subsequent call to **perror()** or **strerror()**. No library function will modify this string.

The **strerror_r()** function is similar to **strerror()**, but is thread safe. It returns the string in the user-supplied buffer *buf* of length *n*.

RETURN VALUE

The **strerror()** function returns the appropriate error description string, or an unknown error message if the error code is unknown. The value of *errno* is not changed for a successful call, and is set to a nonzero value upon error. The **strerror_r()** function returns 0 on success and −1 on failure, setting *errno*.

ERRORS

EINVAL
The value of *errnum* is not a valid error number.

ERANGE
Insufficient storage was supplied to contain the error description string.

CONFORMING TO

SVID 3, POSIX, BSD 4.3, ISO/IEC 9899:1990 (C89).
strerror_r() with prototype as given above is specified by SUSv3, and was in use under Digital Unix and HP Unix. An incompatible function, with prototype

char *strerror_r(int *errnum***, char ***buf***, size_t** *n***);**

is a GNU extension used by glibc (since 2.0), and must be regarded as obsolete in view of SUSv3. The GNU version may, but need not, use the user-supplied buffer. If it does, the result may be truncated in case the supplied buffer is too small. The result is always NUL-terminated.

SEE ALSO

errno(3), **perror**(3), **strsignal**(3)