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

chdir, fchdir − change working directory

SYNOPSIS

#include <unistd.h>

int chdir(const char * *path***);**
int fchdir(int *fd***);**

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

fchdir(): _BSD_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION

chdir() changes the current working directory of the calling process to the directory specified in *path*.

fchdir() is identical to **chdir**(); the only difference is that the directory is given as an open file descriptor.

RETURN VALUE

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

ERRORS

Depending on the file system, other errors can be returned. The more general errors for **chdir**() are listed below:

EACCES
Search permission is denied for one of the components of *path*. (See also **path_resolution**(7).)

EFAULT
path points outside your accessible address space.

EIO An I/O error occurred.

ELOOP
Too many symbolic links were encountered in resolving *path*.

ENAMETOOLONG
path is too long.

ENOENT
The file does not exist.

ENOMEM
Insufficient kernel memory was available.

ENOTDIR
A component of *path* is not a directory.

The general errors for **fchdir**() are listed below:

EACCES
Search permission was denied on the directory open on *fd*.

EBADF
fd is not a valid file descriptor.

CONFORMING TO

SVr4, 4.4BSD, POSIX.1-2001.

SEE ALSO

chroot(2), **getcwd**(3), **path_resolution**(7)

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

fgetc, fgets, getc, getchar, gets, ungetc − input of characters and strings

SYNOPSIS

#include <stdio.h>

int fgetc(FILE * *stream***);**
char *fgets(char * *s***, int** *size***, FILE *** *stream***);**
int getc(FILE * *stream***);**
int getchar(void);
char *gets(char * *s***);**
int ungetc(int *c***, FILE *** *stream***);**

DESCRIPTION

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

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

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

gets() reads a line from *stdin* into the buffer pointed to by *s* until either a terminating newline or **EOF**, which it replaces with **'\0'**. No check for buffer overrun is performed (see **BUGS** below).

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

ungetc() pushes *c* back to *stream*, cast to *unsigned char*, where it is available for subsequent read operations. Pushed-back characters will be returned in reverse order; only one pushback is guaranteed.

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

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

RETURN VALUE

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

gets() and **fgets**() return *s* on success, and NULL on error or when end of file occurs while no characters have been read.

ungetc() returns *c* on success, or **EOF** on error.

CONFORMING TO

C89, C99. LSB deprecates **gets**().

BUGS

Never use **gets**(). Because it is impossible to tell without knowing the data in advance how many characters **gets**() will read, and because **gets**() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use **fgets**() instead.

It is not advisable to mix calls to input functions from the *stdio* library with low-level calls to **read**(2) for the file descriptor associated with the input stream; the results will be undefined and very probably not what you want.

SEE ALSO

read(2), **write**(2), **ferror**(3), **fgetwc**(3), **fgetws**(3), **fopen**(3), **fread**(3), **fseek**(3), **getline**(3), **getwchar**(3), **puts**(3), **scanf**(3), **ungetwc**(3), **unlocked_stdio**(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_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

socket − create an endpoint for communication

SYNOPSIS

#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. The currently understood formats are:

PF_INET ARPA Internet protocols

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

SOCK_STREAM
SOCK_DGRAM

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).

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**. 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.

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.

SEE ALSO

close(2), **read**(2), **write**(2), **accept**(3N), **bind**(3N), **connect**(3N), **listen**(3N),

-(-)

NAME

-