NAME

accept — accept a new connection on a socket

SYNOPSIS

#include <sys/socket.h>

int accept(int *socket*, struct sockaddr *restrict *address*,
socklen_t *restrict *address_len*);

DESCRIPTION

The *accept*() function shall extract the first connection on the queue of pending connections, create a new socket with the same socket type protocol and address family as the specified socket, and allocate a new file descriptor for that socket. The file descriptor shall be allocated as described in *Section 2.14*, *File Descriptor Allocation*.

The *accept*() function takes the following arguments:

socket Specifies a socket that was created with *socket*(), has been bound to an address with *bind*(), and has issued a successful call to *listen*().

address Either a null pointer, or a pointer to a **sockaddr** structure where the address of the connecting socket shall be returned.

address_len Either a null pointer, if *address* is a null pointer, or a pointer to a **socklen_t** object which on input specifies the length of the supplied **sockaddr** structure, and on output specifies the length of the stored address.

If *address* is not a null pointer, the address of the peer for the accepted connection shall be stored in the **sockaddr** structure pointed to by *address*, and the length of this address shall be stored in the object pointed to by *address_len*.

If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the stored address shall be truncated.

If the protocol permits connections by unbound clients, and the peer is not bound, then the value stored in the object pointed to by *address* is unspecified.

If the listen queue is empty of connection requests and O_NONBLOCK is not set on the file descriptor for the socket, *accept*() shall block until a connection is present. If the *listen*() queue is empty of connection requests and O_NONBLOCK is set on the file descriptor for the socket, *accept*() shall fail and set *errno* to **[EAGAIN]** or **[EWOULDBLOCK]**.

The accepted socket cannot itself accept more connections. The original socket remains open and can accept more connections.

RETURN VALUE

Upon successful completion, *accept*() shall return the non-negative file descriptor of the accepted socket. Otherwise, −1 shall be returned, *errno* shall be set to indicate the error, and any object pointed to by *address_len* shall remain unchanged.

ERRORS

The *accept*() function shall fail if:

EAGAIN or **EWOULDBLOCK**
O_NONBLOCK is set for the socket file descriptor and no connections are present to be accepted.

EBADF
The *socket* argument is not a valid file descriptor.

ECONNABORTED
A connection has been aborted.

EINTR
The *accept*() function was interrupted by a signal that was caught before a valid connection arrived.

EINVAL
The *socket* is not accepting connections.

EMFILE
All file descriptors available to the process are currently open.

ENFILE
The maximum number of file descriptors in the system are already open.

ENOBUFS
No buffer space is available.

ENOMEM
There was insufficient memory available to complete the operation.

ENOTSOCK
The *socket* argument does not refer to a socket.

EOPNOTSUPP
The socket type of the specified socket does not support accepting connections.

The *accept*() function may fail if:

EPROTO
A protocol error has occurred; for example, the STREAMS protocol stack has not been initialized.

APPLICATION USAGE

When a connection is available, *select*() indicates that the file descriptor for the socket is ready for reading.

NAME

bind — bind a name to a socket

SYNOPSIS

#include <sys/socket.h>

int bind(int *socket*, const struct sockaddr **address*,
socklen_t *address_len*);

DESCRIPTION

The *bind*() function shall assign a local socket address *address* to a socket identified by descriptor *socket* that has no local socket address assigned. Sockets created with the *socket*() function are initially unnamed; they are identified only by their address family.

The *bind*() function takes the following arguments:

socket Specifies the file descriptor of the socket to be bound.

address Points to a **sockaddr** structure containing the address to be bound to the socket. The length and format of the address depend on the address family of the socket.

address_len Specifies the length of the **sockaddr** structure pointed to by the *address* argument.

The socket specified by *socket* may require the process to have appropriate privileges to use the *bind*() function.

If the address family of the socket is AF_UNIX and the pathname in *address* names a symbolic link, *bind*() shall fail and set *errno* to **[EADDRINUSE]**.

If the socket address cannot be assigned immediately and O_NONBLOCK is set for the file descriptor for the socket, *bind*() shall fail and set *errno* to **[EINPROGRESS]**, but the assignment request shall not be aborted, and the assignment shall be completed asynchronously. Subsequent calls to *bind*() for the same socket, before the assignment is completed, shall fail and set *errno* to **[EALREADY]**.

When the assignment has been performed asynchronously, *pselect*(), *select*(), and *poll*() shall indicate that the file descriptor for the socket is ready for reading and writing.

RETURN VALUE

Upon successful completion, *bind*() shall return 0; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *bind*() function shall fail if:

EADDRINUSE
The specified address is already in use.

EADDRNOTAVAIL
The specified address is not available from the local machine.

EAFNOSUPPORT
The specified address is not a valid address for the address family of the specified socket.

EALREADY
An assignment request is already in progress for the specified socket.

EBADF
The *socket* argument is not a valid file descriptor.

EINPROGRESS
O_NONBLOCK is set for the file descriptor for the socket and the assignment cannot be immediately performed; the assignment shall be performed asynchronously.

EINVAL
The socket is already bound to an address, and the protocol does not support binding to a new address; or the socket has been shut down.

NAME

fclose — close a stream

SYNOPSIS

#include <stdio.h>

int fclose(FILE **stream*);

DESCRIPTION

The *fclose*() function shall cause the stream pointed to by *stream* to be flushed and the associated file to be closed. Any unwritten buffered data for the stream shall be written to the file; any unread buffered data shall be discarded. Whether or not the call succeeds, the stream shall be disassociated from the file and any buffer set by the *setbuf*() or *setvbuf*() function shall be disassociated from the stream. If the associated buffer was automatically allocated, it shall be deallocated.

If the file is not already at EOF, and the file is one capable of seeking, the file offset of the underlying open file description shall be set to the file position of the stream if the stream is the active handle to the underlying file description.

The *fclose*() function shall mark for update the last data modification and last file status change timestamps of the underlying file, if the stream was writable, and if buffered data remains that has not yet been written to the file. The *fclose*() function shall perform the equivalent of a *close*() on the file descriptor that is associated with the stream pointed to by *stream*.

After the call to *fclose*(), any use of *stream* results in undefined behavior.

RETURN VALUE

Upon successful completion, *fclose*() shall return 0; otherwise, it shall return EOF and set *errno* to indicate the error.

ERRORS

The *fclose*() function shall fail if:

EAGAIN
The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the thread would be delayed in the write operation.

EBADF
The file descriptor underlying stream is not valid.

EFBIG
An attempt was made to write a file that exceeds the maximum file size.

EFBIG
An attempt was made to write a file that exceeds the file size limit of the process.

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

EINTR
The *fclose*() function was interrupted by a signal.

ENOMEM
The underlying stream was created by *open_memstream*() or *open_wmemstream*() and insufficient memory is available.

ENOSPC
There was no free space remaining on the device containing the file or in the buffer used by the *fmemopen*() function.

EPIPE An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal shall also be sent to the thread.

NAME

ferror — test error indicator on a stream

SYNOPSIS

#include <stdio.h>

int ferror(FILE *_stream_);

DESCRIPTION

The *ferror*() function shall test the error indicator for the stream pointed to by *stream*.

The *ferror*() function shall not change the setting of *errno* if *stream* is valid.

RETURN VALUE

The *ferror*() function shall return non-zero if and only if the error indicator is set for *stream*.

ERRORS

No errors are defined.

NAME

fgets — get a string from a stream

SYNOPSIS

#include <stdio.h>

char *fgets(char *restrict _s_, int _n_, FILE *restrict _stream_);

DESCRIPTION

The *fgets*() function shall read bytes from *stream* into the array pointed to by *s* until *n*−1 bytes are read, or a <newline> is read and transferred to *s*, or an end-of-file condition is encountered. A null byte shall be written immediately after the last byte read into the array. If the end-of-file condition is encountered before any bytes are read, the contents of the array pointed to by *s* shall not be changed.

The *fgets*() function may mark the last data access timestamp of the file associated with *stream* for update. The last data access timestamp shall be marked for update by the first successful execution of *fgetc*(), *fgets*(), *fread*(), *fscanf*(), *getc*(), *getchar*(), *getdelim*(), *getline*(), *gets*(), or *scanf*() using *stream* that returns data not supplied by a prior call to *ungetc*().

RETURN VALUE

Upon successful completion, *fgets*() shall return *s*. If the stream is at end-of-file, the end-of-file indicator for the stream shall be set and *fgets*() shall return a null pointer. If a read error occurs, the error indicator for the stream shall be set, *fgets*() shall return a null pointer, and shall set *errno* to indicate the error.

ERRORS

The *fgets*() function shall fail if data needs to be read and:

EAGAIN
The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the thread would be delayed in the *fgets*() operation.

EBADF
The file descriptor underlying *stream* is not a valid file descriptor open for reading.

EINTR
The read operation was terminated due to the receipt of a signal, and no data was transferred.

EIO A physical I/O error has occurred, or the process is in a background process group attempting to read from its controlling terminal, and either the calling thread is blocking SIGTTIN or the process is ignoring SIGTTIN or the process group of the process is orphaned. This error may also be generated for implementation-defined reasons.

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.

The *fgets*() function may fail if:

ENOMEM
Insufficient storage space is available.

ENXIO
A request was made of a nonexistent device, or the request was outside the capabilities of the device.

NAME

ipv6 – Linux IPv6 protocol implementation

DESCRIPTION

Linux 2.2 optionally implements the Internet Protocol, version 6. This man page contains a description of the IPv6 basic API as implemented by the Linux kernel and glibc 2.1. The interface is based on the BSD sockets interface; see **socket**(7).

The IPv6 API aims to be mostly compatible with the IPv4 API (see **ip**(7)). Only differences are described in this man page.

To bind an **AF_INET6** socket to any process, the local address should be copied from the *in6addr_any* variable which has *in6_addr* type. In static initializations, **IN6ADDR_ANY_INIT** may also be used, which expands to a constant expression. Both of them are in network byte order.

The IPv6 loopback address (::1) is available in the global *in6addr_loopback* variable. For initializations, **IN6ADDR_LOOPBACK_INIT** should be used.

IPv4 connections can be handled with the v6 API by using the v4-mapped-on-v6 address type; thus a program needs to support only this API type to support both protocols. This is handled transparently by the address handling functions in the C library.

IPv4 and IPv6 share the local port space. When you get an IPv4 connection or packet to an IPv6 socket, its source address will be mapped to v6.

Address format

```
struct sockaddr_in6 {
        uint16_t sin6_family;       /* AF_INET6 */
        uint16_t sin6_port;         /* port number */
        uint32_t sin6_flowinfo;     /* IPv6 flow information */
        struct in6_addr sin6_addr;  /* IPv6 address */
        uint32_t sin6_scope_id;     /* Scope ID (new in 2.4) */
};

struct in6_addr {
        unsigned char   s6_addr[16];    /* IPv6 address */
};
```

sin6_family is always set to **AF_INET6**; *sin6_port* is the protocol port (see *sin_port* in **ip**(7)); *sin6_flowinfo* is the IPv6 flow identifier; *sin6_addr* is the 128-bit IPv6 address. *sin6_scope_id* is an ID depending on the scope of the address. It is new in Linux 2.4. Linux supports it only for link-local addresses, in that case *sin6_scope_id* contains the interface index (see **netdevice**(7))

NAME

listen — listen for socket connections and limit the queue of incoming connections

SYNOPSIS

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

int listen(int socket, int backlog);
```

DESCRIPTION

The *listen*() function shall mark a connection-mode socket, specified by the *socket* argument, as accepting connections.

The *backlog* argument provides a hint to the implementation which the implementation shall use to limit the number of outstanding connections in the socket's listen queue. Implementations may impose a limit on *backlog* and silently reduce the specified value. Normally, a larger *backlog* argument value shall result in a larger or equal length of the listen queue. Implementations shall support values of *backlog* up to SOMAXCONN, defined in <*sys/socket.h*>.

The implementation may include incomplete connections in its listen queue. The limits on the number of incomplete connections and completed connections queued may be different.

The implementation may have an upper limit on the length of the listen queue—either global or per accepting socket. If *backlog* exceeds this limit, the length of the listen queue is set to the limit.

If *listen*() is called with a *backlog* argument value that is less than 0, the function behaves as if it had been called with a *backlog* argument value of 0.

A *backlog* argument of 0 may allow the socket to accept connections, in which case the length of the listen queue may be set to an implementation-defined minimum value.

The socket in use may require the process to have appropriate privileges to use the *listen*() function.

RETURN VALUE

Upon successful completions, *listen*() shall return 0; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *listen*() function shall fail if:

EBADF
The *socket* argument is not a valid file descriptor.

EDESTADDRREQ
The socket is not bound to a local address, and the protocol does not support listening on an unbound socket.

EINVAL
The *socket* is already connected.

ENOTSOCK
The *socket* argument does not refer to a socket.

EOPNOTSUPP
The socket protocol does not support *listen*().

The *listen*() function may fail if:

EACCES
The calling process does not have appropriate privileges.

EINVAL
The *socket* has been shut down.

ENOBUFS
Insufficient resources are available in the system to complete the call.

NAME

pthread_cond_broadcast, pthread_cond_signal — broadcast or signal a condition

SYNOPSIS

#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *_cond_);
int pthread_cond_signal(pthread_cond_t *_cond_);

DESCRIPTION

These functions shall unblock threads blocked on a condition variable.

The *pthread_cond_broadcast*() function shall unblock all threads currently blocked on the specified condition variable *cond*.

The *pthread_cond_signal*() function shall unblock at least one of the threads that are blocked on the specified condition variable *cond* (if any threads are blocked on *cond*).

If more than one thread is blocked on a condition variable, the scheduling policy shall determine the order in which threads are unblocked. When each thread unblocked as a result of a *pthread_cond_broadcast*() or *pthread_cond_signal*() returns from its call to *pthread_cond_wait*() the thread shall own the mutex with which it called *pthread_cond_wait*() The thread(s) that are unblocked shall contend for the mutex according to the scheduling policy (if applicable), and as if each had called *pthread_mutex_lock*().

The *pthread_cond_broadcast*() or *pthread_cond_signal*() functions may be called by a thread whether or not it currently owns the mutex that threads calling *pthread_cond_wait*() have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex shall be locked by the thread calling *pthread_cond_broadcast*() or *pthread_cond_signal*().

The *pthread_cond_broadcast*() and *pthread_cond_signal*() functions shall have no effect if there are no threads currently blocked on *cond*.

The behavior is undefined if the value specified by the *cond* argument to *pthread_cond_broadcast*() or *pthread_cond_signal*() does not refer to an initialized condition variable.

RETURN VALUE

If successful, the *pthread_cond_broadcast*() and *pthread_cond_signal*() functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions shall not return an error code of **[EINTR]**.

NAME

pthread_cond_wait — wait on a condition

SYNOPSIS

#include <pthread.h>

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

DESCRIPTION

The *pthread_cond_wait*() function shall block on a condition variable. The application shall ensure that these functions are called with *mutex* locked by the calling thread; otherwise, undefined behavior results.

These functions atomically release *mutex* and cause the calling thread to block on the condition variable *cond*; atomically here means “atomically with respect to access by another thread to the mutex and then the condition variable”. That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to *pthread_cond_broadcast*() or *pthread_cond_signal*() in that thread shall behave as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex shall have been locked and shall be owned by the calling thread.

When using condition variables there is always a Boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the *pthread_cond_wait*() function may occur. Since the return from *pthread_cond_wait*() does not imply anything about the value of this predicate, the predicate should be re-evaluated upon such return.

When a thread waits on a condition variable, having specified a particular mutex to the *pthread_cond_wait*() operation, a dynamic binding is formed between that mutex and condition variable that remains in effect as long as at least one thread is blocked on the condition variable. During this time, the effect of an attempt by any thread to wait on that condition variable using a different mutex is undefined. Once all waiting threads have been unblocked (as by the *pthread_cond_broadcast*() operation), the next wait operation on that condition variable shall form a new dynamic binding with the mutex specified by that wait operation. Even though the dynamic binding between condition variable and mutex may be removed or replaced between the time a thread is unblocked from a wait on the condition variable and the time that it returns to the caller or begins cancellation cleanup, the unblocked thread shall always re-acquire the mutex specified in the condition wait operation call from which it is returning.

A thread that has been unblocked because it has been canceled while blocked in a call to *pthread_cond_wait*() shall not consume any condition signal that may be directed concurrently at the condition variable if there are other threads blocked on the condition variable.

If a signal is delivered to a thread waiting for a condition variable, upon return from the signal handler the thread resumes waiting for the condition variable as if it was not interrupted, or it shall return zero due to spurious wakeup.

The behavior is undefined if the value specified by the *cond* or *mutex* argument to these functions does not refer to an initialized condition variable or an initialized mutex object, respectively.

RETURN VALUE

Upon successful completion, a value of zero shall be returned; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_cond_wait*() function shall not return an error code of **[EINTR]**.

NAME

pthread_create — thread creation

SYNOPSIS

#include <pthread.h>

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

DESCRIPTION

The *pthread_create*() function shall create a new thread, with attributes specified by *attr*, within a process. If *attr* is NULL, the default attributes shall be used. If the attributes specified by *attr* are modified later, the thread's attributes shall not be affected. Upon successful completion, *pthread_create*() shall store the ID of the created thread in the location referenced by *thread*.

The thread is created executing *start_routine* with *arg* as its sole argument. If the *start_routine* returns, the effect shall be as if there was an implicit call to *pthread_exit*() using the return value of *start_routine* as the exit status. Note that the thread in which *main*() was originally invoked differs from this. When it returns from *main*(), the effect shall be as if there was an implicit call to *exit*() using the return value of *main*() as the exit status.

The signal state of the new thread shall be initialized as follows:

* The signal mask shall be inherited from the creating thread.
* The set of signals pending for the new thread shall be empty.

The thread-local current locale and the alternate stack shall not be inherited.

The floating-point environment shall be inherited from the creating thread.

If *pthread_create*() fails, no new thread is created and the contents of the location referenced by *thread* are undefined.

If _POSIX_THREAD_CPUTIME is defined, the new thread shall have a CPU-time clock accessible, and the initial value of this clock shall be set to zero.

The behavior is undefined if the value specified by the *attr* argument to *pthread_create*() does not refer to an initialized thread attributes object.

RETURN VALUE

If successful, the *pthread_create*() function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_create*() function shall fail if:

EAGAIN
The system lacked the necessary resources to create another thread, or the system-imposed limit on the total number of threads in a process {PTHREAD_THREADS_MAX} would be exceeded.

EPERM
The caller does not have appropriate privileges to set the required scheduling parameters or scheduling policy.

The *pthread_create*() function shall not return an error code of **[EINTR]**.

NAME

pthread_mutex_lock, pthread_mutex_unlock — lock and unlock a mutex

SYNOPSIS

#include <pthread.h>

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

DESCRIPTION

The mutex object referenced by *mutex* shall be locked by a call to *pthread_mutex_lock*() that returns zero. If the mutex is already locked by another thread, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by *mutex* in the locked state with the calling thread as its owner.

The *pthread_mutex_unlock*() function shall release the mutex object referenced by *mutex*. If there are threads blocked on the mutex object referenced by *mutex* when *pthread_mutex_unlock*() is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread shall resume waiting for the mutex as if it was not interrupted.

If *mutex* does not refer to an initialized mutex object, the behavior of *pthread_mutex_lock*() and *pthread_mutex_unlock*() is undefined.

RETURN VALUE

If successful, the *pthread_mutex_lock*() and *pthread_mutex_unlock*() functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutex_lock*() function shall fail if:

EINVAL
The *mutex* was created with the protocol attribute having the value PTHREAD_PRIO_PROTECT and the calling thread's priority is higher than the mutex's current priority ceiling.

ENOTRECOVERABLE
The state protected by the mutex is not recoverable.

EDEADLK
The mutex type is PTHREAD_MUTEX_ERRORCHECK and the current thread already owns the mutex.

The *pthread_mutex_unlock*() function shall fail if:

EPERM
The mutex type is PTHREAD_MUTEX_ERRORCHECK or PTHREAD_MUTEX_RECURSIVE, or the mutex is a robust mutex, and the current thread does not own the mutex.

The *pthread_mutex_lock*() function may fail if:

EDEADLK
A deadlock condition was detected.

These functions shall not return an error code of **[EINTR]**.

NAME

realloc — memory reallocator

SYNOPSIS

#include <stdlib.h>

void *realloc(void *_ptr_, size_t _size_);

DESCRIPTION

The *realloc*() function shall deallocate the old object pointed to by *ptr* and return a pointer to a new object that has the size specified by *size*. The contents of the new object shall be the same as that of the old object prior to deallocation, up to the lesser of the new and old sizes. Any bytes in the new object beyond the size of the old object have indeterminate values. If the size of the space requested is zero, the behavior shall be implementation-defined: either a null pointer is returned, or the behavior shall be as if the size were some non-zero value, except that the behavior is undefined if the returned pointer is used to access an object. If the space cannot be allocated, the object shall remain unchanged.

If *ptr* is a null pointer, *realloc*() shall be equivalent to *malloc*() for the specified size.

If *ptr* does not match a pointer returned earlier by *calloc*(), *malloc*(), or *realloc*() or if the space has previously been deallocated by a call to *free*() or *realloc*(), the behavior is undefined.

The order and contiguity of storage allocated by successive calls to *realloc*() is unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned shall point to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall be returned.

RETURN VALUE

Upon successful completion, *realloc*() shall return a pointer to the (possibly moved) allocated space. If *size* is 0, either:

* A null pointer shall be returned and, if *ptr* is not a null pointer, *errno* shall be set to an implementation-defined value.
* A pointer to the allocated space shall be returned, and the memory object pointed to by *ptr* shall be freed. The application shall ensure that the pointer is not used to access an object.

If there is not enough available memory, *realloc*() shall return a null pointer and set *errno* to **[ENOMEM]**. If *realloc*() returns a null pointer and *errno* has been set to **[ENOMEM]**, the memory referenced by *ptr* shall not be changed.

ERRORS

The *realloc*() function shall fail if:

ENOMEM
Insufficient memory is available.

NAME

sigaction — examine and change a signal action

SYNOPSIS

#include <signal.h>

int sigaction(int *sig*, const struct sigaction *restrict *act*,
struct sigaction *restrict *oldact*);

DESCRIPTION

The *sigaction*() function allows the calling process to examine and/or specify the action to be associated with a specific signal. The argument *sig* specifies the signal; acceptable values are defined in <*signal.h*>.

The structure **sigaction**, used to describe an action to be taken, is defined in the <*signal.h*> header to include at least the following members:

Member Type	Member Name	Description
void(*) (int)	*sa_handler*	Pointer to a signal-catching function or one of the macros SIG_IGN or SIG_DFL.
sigset_t	*sa_mask*	Additional set of signals to be blocked during execution of signal-catching function.
int	*sa_flags*	Special flags to affect behavior of signal.

If the argument *act* is not a null pointer, it points to a structure specifying the action to be associated with the specified signal. If the argument *oldact* is not a null pointer, the action previously associated with the signal is stored in the location pointed to by the argument *oldact*. If the argument *act* is a null pointer, signal handling is unchanged; thus, the call can be used to enquire about the current handling of a given signal. The SIGKILL and SIGSTOP signals shall not be added to the signal mask using this mechanism; this restriction shall be enforced by the system without causing an error to be indicated.

The *sa_flags* field can be used to modify the behavior of the specified signal.

The following flags, defined in the <*signal.h*> header, can be set in *sa_flags*:

SA_NOCLDSTOP
Do not generate SIGCHLD when children stop or stopped children continue.

If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is not set in *sa_flags*, and the implementation supports the SIGCHLD signal, then a SIGCHLD signal shall be generated for the calling process whenever any of its child processes stop and a SIGCHLD signal may be generated for the calling process whenever any of its stopped child processes are continued. If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is set in *sa_flags*, then the implementation shall not generate a SIGCHLD signal in this way.

SA_RESTART This flag affects the behavior of interruptible functions; that is, those specified to fail with *errno* set to **[EINTR]**. If set, and a function specified as interruptible is interrupted by this signal, the function shall restart and shall not fail with **[EINTR]** unless otherwise specified. If an interruptible function which uses a timeout is restarted, the duration of the timeout following the restart is set to an unspecified value that does not exceed the original timeout value. If the flag is not set, interruptible functions interrupted by this signal shall fail with *errno* set to **[EINTR]**.

When a signal is caught by a signal-catching function installed by *sigaction*(), a new signal mask is calculated and installed for the duration of the signal-catching function (or until a call to either *sigprocmask*() or *sigsuspend*() is made). This mask is formed by taking the union of the current signal mask and the value of the *sa_mask* for the signal being delivered, and includes the signal being delivered. If and when the user's signal handler returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it shall remain installed until another action is explicitly requested (by another call to *sigaction*()), until one of the *exec* functions is called.

If the previous action for *sig* had been established by *signal*(), the values of the fields returned in the structure pointed to by *oldact* are unspecified, and in particular *oldact*->*sa_handler* is not necessarily the same

value passed to *signal*(). However, if a pointer to the same structure or a copy thereof is passed to a subsequent call to *sigaction*() via the *act* argument, handling of the signal shall be as if the original call to *signal*() were repeated.

If *sigaction*() fails, no new signal handler is installed.

It is unspecified whether an attempt to set the action for a signal that cannot be caught or ignored to SIG_DFL is ignored or causes an error to be returned with *errno* set to **[EINVAL]**.

The result of the use of *sigaction*() and a *sigwait*() function concurrently within a process on the same signal is unspecified.

RETURN VALUE

Upon successful completion, *sigaction*() shall return 0; otherwise, −1 shall be returned, *errno* shall be set to indicate the error, and no new signal-catching function shall be installed.

ERRORS

The *sigaction*() function shall fail if:

EINVAL
The *sig* argument is not a valid signal number or an attempt is made to catch a signal that cannot be caught or ignore a signal that cannot be ignored.

The *sigaction*() function may fail if:

EINVAL
An attempt was made to set the action to SIG_DFL for a signal that cannot be caught or ignored (or both).

NAME

socket — create an endpoint for communication

SYNOPSIS

#include <sys/socket.h>

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

DESCRIPTION

The *socket*() function shall create an unbound socket in a communications domain, and return a file descriptor that can be used in later function calls that operate on sockets. The file descriptor shall be allocated as described in *Section 2.14*, *File Descriptor Allocation*.

The *socket*() function takes the following arguments:

domain Specifies the communications domain in which a socket is to be created.

type Specifies the type of socket to be created.

protocol Specifies a particular protocol to be used with the socket. Specifying a *protocol* of 0 causes *socket*() to use an unspecified default protocol appropriate for the requested socket type.

The *domain* argument specifies the address family used in the communications domain. The address families supported by the system are implementation-defined.

Symbolic constants that can be used for the domain argument are defined in the *<sys/socket.h>* header.

The *type* argument specifies the socket type, which determines the semantics of communication over the socket. The following socket types are defined; implementations may specify additional socket types:

SOCK_STREAM
Provides sequenced, reliable, bidirectional, connection-mode byte streams, and may provide a transmission mechanism for out-of-band data.

SOCK_DGRAM

SOCK_SEQPACKET

If the *protocol* argument is non-zero, it shall specify a protocol that is supported by the address family. If the *protocol* argument is zero, the default protocol for this address family and type shall be used. The protocols supported by the system are implementation-defined.

The process may need to have appropriate privileges to use the *socket*() function or to create some sockets.

RETURN VALUE

Upon successful completion, *socket*() shall return a non-negative integer, the socket file descriptor. Otherwise, a value of −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *socket*() function shall fail if:

EAFNOSUPPORT
The implementation does not support the specified address family.

EMFILE
All file descriptors available to the process are currently open.

ENFILE
No more file descriptors are available for the system.

EPROTONOSUPPORT
The protocol is not supported by the address family, or the protocol is not supported by the implementation.

EPROTOTYPE
The socket type is not supported by the protocol.