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, fileno − stream open functions

SYNOPSIS

#include <stdio.h>

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

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.

The function **fileno**() examines the argument *stream* and returns its integer descriptor.

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, fputc, fputs, putc, putchar − input and output 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);
int fputc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

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***)**.

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.

fputc() writes the character *c*, cast to an *unsigned char*, to *stream*.

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

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

putchar(*c***);** is equivalent to **putc(***c***,** *stdout***)**.

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

RETURN VALUE

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

fgets() returns *s* on success, and NULL on error or when end of file occurs while no characters have been read. **fputc**(), **putc**() and **putchar**() return the character written as an *unsigned char* cast to an *int* or **EOF** on error.

fputs() returns a nonnegative number on success, or **EOF** on error.

SEE ALSO

read(2), **write**(2), **ferror**(3), **fgetwc**(3), **fgetws**(3), **fopen**(3), **fread**(3), **fseek**(3), **getline**(3), **getwchar**(3), **scanf**(3), **ungetwc**(3), **write**(2), **ferror**(3), **fopen**(3), **fputwc**(3), **fputws**(3), **fseek**(3), **fwrite**(3), **gets**(3), **putwchar**(3), **scanf**(3), **unlocked_stdio**(3)

NAME

ipv6, PF_INET6 − Linux IPv6 protocol implementation

SYNOPSIS

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

tcp6_socket = socket(PF_INET6, SOCK_STREAM, 0);
raw6_socket = socket(PF_INET6, SOCK_RAW, protocol);
udp6_socket = socket(PF_INET6, SOCK_DGRAM, protocol);
```

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 **ip**(7) v4 API. 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 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 only needs only to support this API type to support both protocols. This is handled transparently by the address handling functions in libc.

IPv4 and IPv6 share the local port space. When you get an IPv4 connection or packet to a IPv6 socket its source address will be mapped to v6 and it 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 of depending of on the scope of the address. It is new in Linux 2.4. Linux only supports it for link scope addresses, in that case *sin6_scope_id* contains the interface index (see **netdevice**(7))

NOTES

The *sockaddr_in6* structure is bigger than the generic *sockaddr*. Programs that assume that all address types can be stored safely in a *struct sockaddr* need to be changed to use *struct sockaddr_storage* for that instead.

SEE ALSO

cmsg(3), **ip**(7)

NAME

listen − listen for connections on a socket

SYNOPSIS

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

int listen(int *sockfd***, int** *backlog***);**

DESCRIPTION

listen() marks the socket referred to by *sockfd* as a passive socket, that is, as a socket that will be used to accept incoming connection requests using **accept**(2).

The *sockfd* argument is a file descriptor that refers to a socket of type **SOCK_STREAM** or **SOCK_SEQPACKET**.

The *backlog* argument defines the maximum length to which the queue of pending connections for *sockfd* may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of **ECONNREFUSED** or, if the underlying protocol supports retransmission, the request may be ignored so that a later reattempt at connection succeeds.

RETURN VALUE

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

ERRORS

EADDRINUSE
Another socket is already listening on the same port.

EBADF
The argument *sockfd* is not a valid descriptor.

ENOTSOCK
The argument *sockfd* is not a socket.

NOTES

To accept connections, the following steps are performed:

1. A socket is created with **socket**(2).
2. The socket is bound to a local address using **bind**(2), so that other sockets may be **connect**(2)ed to it.
3. A willingness to accept incoming connections and a queue limit for incoming connections are specified with **listen**().
4. Connections are accepted with **accept**(2).

If the *backlog* argument is greater than the value in */proc/sys/net/core/somaxconn*, then it is silently truncated to that value; the default value in this file is 128.

EXAMPLE

See **bind**(2).

SEE ALSO

accept(2), **bind**(2), **connect**(2), **socket**(2), **socket**(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 / pthread_exit − terminate the calling thread

SYNOPSIS

#include <pthread.h>

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

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

DESCRIPTION

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

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

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

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

RETURN VALUE

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

The **pthread_exit** function never returns.

ERRORS

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

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

AUTHOR

Xavier Leroy <Xavier.Leroy@inria.fr>

SEE ALSO

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

NAME

pthread_detach − put a running thread in the detached state

SYNOPSIS

#include <pthread.h>

int pthread_detach(pthread_t th);

DESCRIPTION

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

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

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

RETURN VALUE

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

ERRORS

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

EINVAL
the thread *th* is already in the detached state

AUTHOR

Xavier Leroy <Xavier.Leroy@inria.fr>

SEE ALSO

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

NAME

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

SYNOPSIS

#include <pthread.h>

pthread_mutex_t *fastmutex* = **PTHREAD_MUTEX_INITIALIZER;**

pthread_mutex_t *recmutex* = **PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;**

pthread_mutex_t *errchkmutex* = **PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;**

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

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

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

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

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

DESCRIPTION

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

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

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

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

Variables of type **pthread_mutex_t** can also be initialized statically, using the constants **PTHREAD_MUTEX_INITIALIZER** (for fast mutexes), **PTHREAD_RECURSIVE_MUTEX_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

stat, fstat, lstat − get file status

SYNOPSIS

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

int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

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

lstat(): _BSD_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION

These functions return information about a file. No permissions are required on the file itself, but — in the case of **stat**() and **lstat**() — execute (search) permission is required on all of the directories in *path* that lead to the file.

stat() stats the file pointed to by *path* and fills in *buf* .

lstat() is identical to **stat**(), except that if *path* is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

fstat() is identical to **stat**(), except that the file to be stat-ed is specified by the file descriptor *fd*.

All of these system calls return a *stat* structure, which contains the following fields:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    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 ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system 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 *st_dev* field describes the device on which this file resides.

The *st_rdev* field describes the device that this file (inode) represents.

The *st_size* field gives the size of the file (if it is a regular file or a symbolic link) in bytes. The size of a symlink is the length of the pathname it contains, without a trailing null byte.

The *st_blocks* field indicates the number of blocks allocated to the file, 512-byte units. (This may be smaller than *st_size*/512 when the file has holes.)

The *st_blksize* field 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 file systems 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, for example, 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, for example, 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 using the *st_mode* field:

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 (named pipe)?
S_ISLNK(m)	symbolic link? (Not in POSIX.1-1996.)
S_ISSOCK(m)	socket? (Not in POSIX.1-1996.)

RETURN VALUE

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

ERRORS

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

EBADF
fd is bad.

EFAULT
Bad address.

ELOOP
Too many symbolic links encountered while traversing the path.

ENAMETOOLONG
File name too long.

ENOENT
A component of the path *path* does not exist, or the path is an empty string.

ENOMEM
Out of memory (i.e., kernel memory).

ENOTDIR
A component of the path is not a directory.

SEE ALSO

access(2), **chmod**(2), **chown**(2), **fstatat**(2), **readlink**(2), **utime**(2), **capabilities**(7), **symlink**(7)