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 VALUE

On success, these system calls return a nonnegative integer that is a file descriptor for the accepted socket. On error, −1 is returned, and *errno* is set appropriately.

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

bbCreate, bbPut, bbGet, bbDestroy – A synchronized bounded-buffer implementation

SYNOPSIS

#include ”bbuffer.h”

BNDBUF *bbCreate(size_t *size*);
void bbPut(BNDBUF * *bb*, **int** *value*);
int bbGet(BNDBUF * *bb*);
void bbDestroy(BNDBUF * *bb*);

DESCRIPTION

Bounded-buffer implementation of a FIFO queue. Manages **int** and supports multiple concurrent rea and writers. Provides the following functions:

bbCreate() creates a new bounded buffer for up to *size* elements. If an error occurs during the initial tion, the implementation frees all resources already allocated by then and returns **NULL.**

bbPut() stores the *value* in the bounded buffer. If the buffer is full (i.e., it currently contains *size* eleme the call to **bbPut**() blocks until the value can be stored.

bbGet() returns the next value from the bounded buffer. If the buffer is empty, the call blocks until a v is available.

Both **bbPut**() and **bbGet**() are synchronized internally and thus can be called concurrently without the n for further synchronization.

bbDestroy() releases any resources related to the bounded buffer itself. It does not call free() on the ments stored in the buffer.

RETURN VALUE

bbCreate() returns a pointer to the allocated bounded buffer, or **NULL** if the request fails.

bbPut() returns no value.

bbGet() returns the next value stored in the bounded buffer.

bbDestroy() returns no value.

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 *s* . 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 VALUE

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

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

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*);
int fclose(FILE **stream*);

DESCRIPTION

The **fopen** function opens the file whose name is the string pointed to by *path* and associates a stream w it.

The argument *mode* points to a string beginning with one of the following sequences (Additional charac 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 beginn of the file.

w+ Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. 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 strea 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 ex 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 str (one of the values "r", "r+", "w", "w+", "a", "a+") must be compatible with the mode of the file descrip The file position indicator of the new stream is set to that belonging to *fildes*, and the error and end-of indicators are cleared. Modes "w" or "w+" do not cause truncation of the file. The file descriptor is dup'ed, and will be closed when the stream created by **fdopen** is closed. The result of applying **fdopen** shared memory object is undefined.

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

The **fclose**() function flushes the stream pointed to by *stream* (writing any buffered output data us **fflush**(3)) and closes the underlying file descriptor.

RETURN VALUE

Upon successful completion **fopen**, **fdopen** and **freopen** return a **FILE** pointer. Otherwise, **NULL** returned and the global variable *errno* is set to indicate the error. Upon successful completion of **fclos** is returned. Otherwise, **EOF** is returned and *errno* is set to indicate the error.

ERRORS

EINVAL

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

EBADF

The file descriptor underlying *stream* passed to **fclose** is not valid.

The **fopen**, **fdopen** and **freopen** functions may also fail and set *errno* for any of the errors specified for 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).

NAME

ipv6, AF_INET6 – Linux IPv6 protocol implementation

SYNOPSIS

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

tcp6_socket **= socket(AF_INET6, SOCK_STREAM, 0);**
raw6_socket **= socket(AF_INET6, SOCK_RAW,** *protocol***);**
udp6_socket **= socket(AF_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.

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))

RETURN VALUES

−1 is returned if an error occurs. Otherwise the return value is a descriptor referencing the socket.

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 use 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_SE PACKET**.

The *backlog* argument defines the maximum length to which the queue of pending connections for *so* may grow. If a connection request arrives when the queue is full, the client may receive an error with indication of **ECONNREFUSED** or, if the underlying protocol supports retransmission, the request may 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 to it.
3. A willingness to accept incoming connections and a queue limit for incoming connections 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 t cated 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

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_join – join with a terminated thread

SYNOPSIS

#include <pthread.h>

int pthread_join(pthread_t *thread*, **void** ***retval*);

Compile and link with *–pthread.*

DESCRIPTION

The **pthread_join**() function waits for the thread specified by *thread* to terminate. If that thread already terminated, then **pthread_join**() returns immediately. The thread specified by *thread* must be able.

If *retval* is not NULL, then **pthread_join**() copies the exit status of the target thread (i.e., the value that target thread supplied to **pthread_exit**(3)) into the location pointed to by *retval*. If the target thread canceled, then **PTHREAD_CANCELED** is placed in the location pointed to by *retval*.

If multiple threads simultaneously try to join with the same thread, the results are undefined. If the th calling **pthread_join**() is canceled, then the target thread will remain joinable (i.e., it will not be detache

RETURN VALUE

On success, **pthread_join**() returns 0; on error, it returns an error number.

ERRORS

EDEADLK
A deadlock was detected (e.g., two threads tried to join with each other); or *thread* specifies calling thread.

EINVAL
thread is not a joinable thread.

EINVAL
Another thread is already waiting to join with this thread.

ESRCH
No thread with the ID *thread* could be found.

NOTES

After a successful call to **pthread_join**(), the caller is guaranteed that the target thread has terminated. caller may then choose to do any clean-up that is required after termination of the thread (e.g., fre memory or other resources that were allocated to the target thread).

Joining with a thread that has previously been joined results in undefined behavior.

Failure to join with a thread that is joinable (i.e., one that is not detached), produces a "zombie thre Avoid doing this, since each zombie thread consumes some system resources, and when enough zon threads have accumulated, it will no longer be possible to create new threads (or processes).

There is no pthreads analog of *waitpid(-1, &status, 0)*, that is, "join with any terminated thread". If believe you need this functionality, you probably need to rethink your application design.

All of the threads in a process are peers: any thread can join with any other thread in the process.

EXAMPLE

See **pthread_create**(3).

SEE ALSO

pthread_cancel(3), **pthread_create**(3), **pthread_detach**(3), **pthread_exit**(3), **pthreads**(7)

NAME

sigaction – POSIX signal handling functions.

SYNOPSIS

#include <signal.h>

int sigaction(int *signum*, **const struct sigaction** **act*, **struct sigaction** **oldact*);

DESCRIPTION

The **sigaction** system call is used to change the action taken by a process on receipt of a specific signal.

signum specifies the signal and can be any valid signal except **SIGKILL** and **SIGSTOP**.

If *act* is non−null, the new action for signal *signum* is installed from *act*. If *oldact* is non−null, the previous action is saved in *oldact*.

The **sigaction** structure is defined as something like

```
struct sigaction {
    void     (*sa_handler)(int signal_number);
    sigset_t sa_mask;
    int      sa_flags;
}
```

sa_handler specifies the action to be associated with *signum* and may be **SIG_DFL** for the default action, **SIG_IGN** to ignore this signal, or a pointer to a signal handling function.

sa_mask gives a mask of signals which should be blocked during execution of the signal handler. In addition, the signal which triggered the handler will be blocked, unless the **SA_NODEFER** or **SA_NOMASK** flags are used.

sa_flags specifies a set of flags which modify the behaviour of the signal handling process. It is formed by the bitwise OR of zero or more of the following:

SA_NOCLDSTOP
: If *signum* is **SIGCHLD**, do not receive notification when child processes stop (i.e., when child processes receive one of **SIGSTOP**, **SIGTSTP**, **SIGTTIN** or **SIGTTOU**).

SA_RESTART
: Provide behaviour compatible with BSD signal semantics by making certain system calls restartable across signals. Without **SA_RESTART** the system calls return an error and set errno to EINTR when interrupted by a signal.

RETURN VALUES

sigaction() returns 0 on success; on error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINVAL
: An invalid signal was specified. This will also be generated if an attempt is made to change the action for **SIGKILL** or **SIGSTOP**, which cannot be caught.

SEE ALSO

kill(1), **kill**(2), **killpg**(2), **pause**(2), **sigsetops**(3),

NAME

sigprocmask – change and/or examine caller's signal mask
sigsuspend – install a signal mask and suspend caller until signal

SYNOPSIS

#include <signal.h>

int sigprocmask(int *how*, **const sigset_t** **set*, **sigset_t** **oset*);

int sigsuspend(const sigset_t **set*);

DESCRIPTION sigprocmask

The **sigprocmask**( ) function is used to examine and/or change the caller's signal mask. If the valu **SIG_BLOCK**, the set pointed to by the argument *set* is added to the current signal mask. If the valu **SIG_UNBLOCK**, the set pointed by the argument *set* is removed from the current signal mask. If the va is **SIG_SETMASK**, the current signal mask is replaced by the set pointed to by the argument *set*. If argument *oset* is not NULL, the previous mask is stored in the space pointed to by *oset*. If the value of argument *set* is NULL, the value *how* is not significant and the caller's signal mask is unchanged; thus, call can be used to inquire about currently blocked signals.

If there are any pending unblocked signals after the call to **sigprocmask**( ), at least one of those signals be delivered before the call to **sigprocmask**( ) returns.

It is not possible to block those signals that cannot be ignored this restriction is silently imposed by the tem. See **sigaction**(2).

If **sigprocmask**( ) fails, the caller's signal mask is not changed.

RETURN VALUES

On success, **sigprocmask**( ) returns **0**. On failure, it returns **−1** and sets **errno** to indicate the error.

ERRORS

sigprocmask( ) fails if any of the following is true:

EFAULT *set* or *oset* points to an illegal address.

EINVAL The value of the *how* argument is not equal to one of the defined values.

DESCRIPTION sigsuspend

sigsuspend( ) replaces the caller's signal mask with the set of signals pointed to by the argument *set* then suspends the caller until delivery of a signal whose action is either to execute a signal catching fu tion or to terminate the process.

If the action is to terminate the process, **sigsuspend**( ) does not return. If the action is to execute a sig catching function, **sigsuspend**( ) returns after the signal catching function returns. On return, the sig mask is restored to the set that existed before the call to **sigsuspend**( ).

It is not possible to block those signals that cannot be ignored (see **signal**(5)); this restriction is sile imposed by the system.

RETURN VALUES

Since **sigsuspend**( ) suspends process execution indefinitely, there is no successful completion return va On failure, it returns −1 and sets **errno** to indicate the error.

ERRORS

sigsuspend( ) fails if either of the following is true:

EFAULT *set* points to an illegal address.

EINTR A signal is caught by the calling process and control is returned from the signal catch function.

SEE ALSO

sigaction(2), **sigsetops**(3C),

NAME

sigsetops, sigemptyset, sigfillset, sigaddset, sigdelset, sigismember – manipulate sets of signals

SYNOPSIS

#include <signal.h>

int sigemptyset(sigset_t *_set_);

int sigfillset(sigset_t *_set_);

int sigaddset(sigset_t *_set_, int _signo_);

int sigdelset(sigset_t *_set_, int _signo_);

int sigismember(sigset_t *_set_, int _signo_);

DESCRIPTION

These functions manipulate *sigset_t* data types, representing the set of signals supported by the implementation.

sigemptyset( ) initializes the set pointed to by *set* to exclude all signals defined by the system.

sigfillset( ) initializes the set pointed to by *set* to include all signals defined by the system.

sigaddset( ) adds the individual signal specified by the value of *signo* to the set pointed to by *set*.

sigdelset( ) deletes the individual signal specified by the value of *signo* from the set pointed to by *set*.

sigismember( ) checks whether the signal specified by the value of *signo* is a member of the set pointed to by *set*.

Any object of type *sigset_t* must be initialized by applying either **sigemptyset( )** or **sigfillset( )** before applying any other operation.

RETURN VALUES

Upon successful completion, the **sigismember( )** function returns a value of one if the specified signal is a member of the specified set, or a value of 0 if it is not. Upon successful completion, the other functions return a value of 0. Otherwise a value of –1 is returned and **errno** is set to indicate the error.

ERRORS

sigaddset( ), **sigdelset( )**, and **sigismember( )** will fail if the following is true:

EINVAL The value of the *signo* argument is not a valid signal number.

sigfillset( ) will fail if the following is true:

EFAULT The *set* argument specifies an invalid address.

SEE ALSO

sigaction(2), **sigpending**(2), **sigprocmask**(2), **sigsuspend**(2), **attributes**(5), **signal**(5)

NAME

strtok, strtok_r – extract tokens from strings

SYNOPSIS

#include <string.h>

char *strtok(char *_str_, const char *_delim_);

char *strtok_r(char *_str_, const char *_delim_, char **_saveptr_);

DESCRIPTION

The **strtok**() function breaks a string into a sequence of zero or more nonempty tokens. On the first ca
strtok() the string to be parsed should be specified in *str*. In each subsequent call that should parse
same string, *str* must be NULL.

The *delim* argument specifies a set of bytes that delimit the tokens in the parsed string. The caller
specify different strings in *delim* in successive calls that parse the same string.

Each call to **strtok**() returns a pointer to a null-terminated string containing the next token. This string
not include the delimiting byte. If no more tokens are found, **strtok**() returns NULL.

A sequence of calls to **strtok**() that operate on the same string maintains a pointer that determines the p
from which to start searching for the next token. The first call to **strtok**() sets this pointer to point to
first byte of the string. The start of the next token is determined by scanning forward for the next nondel
iter byte in *str*. If such a byte is found, it is taken as the start of the next token. If no such byte is fou
then there are no more tokens, and **strtok**() returns NULL. (A string that is empty or that contains
delimiters will thus cause **strtok**() to return NULL on the first call.)

The end of each token is found by scanning forward until either the next delimiter byte is found or until
terminating null byte ('\0') is encountered. If a delimiter byte is found, it is overwritten with a null byt
terminate the current token, and **strtok**() saves a pointer to the following byte; that pointer will be use
the starting point when searching for the next token. In this case, **strtok**() returns a pointer to the sta
the found token.

From the above description, it follows that a sequence of two or more contiguous delimiter bytes in
parsed string is considered to be a single delimiter, and that delimiter bytes at the start or end of the st
are ignored. Put another way: the tokens returned by **strtok**() are always nonempty strings. Thus,
example, given the string "*aaa;;bbb,*", successive calls to **strtok**() that specify the delimiter string
would return the strings "*aaa*" and "*bbb*", and then a null pointer.

The **strtok_r**() function is a reentrant version **strtok**(). The *saveptr* argument is a pointer to a *char* *
able that is used internally by **strtok_r**() in order to maintain context between successive calls that parse
same string. On the first call to **strtok_r**(), *str* should point to the string to be parsed, and the valu
saveptr is ignored. In subsequent calls, *str* should be NULL, and *saveptr* should be unchanged since
previous call.

Different strings may be parsed concurrently using sequences of calls to **strtok_r**() that specify diffe
saveptr arguments.

RETURN VALUE

strtok() and **strtok_r**() return a pointer to the next token, or NULL if there are no more tokens.

ATTRIBUTES

Multithreading (see pthreads(7))

The **strtok**() function is not thread-safe, the **strtok_r**() function is thread-safe.