NAME

accept – accept a connection on a socket

SYNOPSIS

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

int accept(int *s***, struct sockaddr** **addr***, int** **addrlen***);**

DESCRIPTION

The argument *s* is a socket that has been created with **socket**(3N) and bound to an address with **bind**(3N), and that is listening for connections after a call to **listen**(3N). The **accept**() function extracts the first connection on the queue of pending connections, creates a new socket with the properties of *s*, and allocates a new file descriptor, *ns*, for the socket. If no pending connections are present on the queue and the socket is not marked as non-blocking, **accept**() blocks the caller until a connection is present. If the socket is marked as non-blocking and no pending connections are present on the queue, **accept**() returns an error as described below. The **accept**() function uses the **netconfig**(4) file to determine the STREAMS device file name associated with *s*. This is the device on which the connect indication will be accepted. The accepted socket, *ns*, is used to read and write data to and from the socket that connected to *ns*; it is not used to accept more connections. The original socket (*s*) remains open for accepting further connections.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity as it is known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication occurs.

The argument *addrlen* is a value-result parameter. Initially, it contains the amount of space pointed to by *addr*; on return it contains the length in bytes of the address returned.

The **accept**() function is used with connection-based socket types, currently with **SOCK_STREAM**.

It is possible to **select**(3C) or **poll**(2) a socket for the purpose of an **accept**() by selecting or polling it for a read. However, this will only indicate when a connect indication is pending; it is still necessary to call **accept**().

RETURN VALUES

The **accept**() function returns **−1** on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

ERRORS

accept() will fail if:

EBADF The descriptor is invalid.

EINTR The accept attempt was interrupted by the delivery of a signal.

EMFILE The per-process descriptor table is full.

ENODEV The protocol family and type corresponding to *s* could not be found in the **netconfig** file.

ENOMEM There was insufficient user memory available to complete the operation.

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

EWOULDBLOCK The socket is marked as non-blocking and no connections are present to be accepted.

SEE ALSO

poll(2), **bind**(3N), **connect**(3N), **listen**(3N), **select**(3C), **socket**(3N), **netconfig**(4), **attributes**(5), **socket**(5)

NAME

bind – bind a name to a socket

SYNOPSIS

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

int bind(int *s***, const struct sockaddr** **name***, int** *namelen***);**

DESCRIPTION

bind() assigns a name to an unnamed socket. When a socket is created with **socket**(3N), it exists in a name space (address family) but has no name assigned. **bind**() requests that the name pointed to by *name* be assigned to the socket.

RETURN VALUES

If the bind is successful, **0** is returned. A return value of **−1** indicates an error, which is further specified in the global **errno**.

ERRORS

The **bind**() call will fail if:

EACCES The requested address is protected and the current user has inadequate permission to access it.

EADDRINUSE The specified address is already in use.

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

EBADF *s* is not a valid descriptor.

EINVAL *namelen* is not the size of a valid address for the specified address family.

EINVAL The socket is already bound to an address.

ENOSR There were insufficient STREAMS resources for the operation to complete.

ENOTSOCK *s* is a descriptor for a file, not a socket.

The following errors are specific to binding names in the UNIX domain:

EACCES Search permission is denied for a component of the path prefix of the pathname in *name*.

EIO An I/O error occurred while making the directory entry or allocating the inode.

EISDIR A null pathname was specified.

ELOOP Too many symbolic links were encountered in translating the pathname in *name*.

ENOENT A component of the path prefix of the pathname in *name* does not exist.

ENOTDIR A component of the path prefix of the pathname in *name* is not a directory.

EROFS The inode would reside on a read-only file system.

SEE ALSO

unlink(2), **socket**(3N), **attributes**(5), **socket**(5)

NOTES

Binding a name in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed (using **unlink**(2)).

The rules used in name binding vary between communication domains.

NAME

chdir, fchdir – change working directory

SYNOPSIS

#include <unistd.h>

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

DESCRIPTION

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

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

RETURN VALUE

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

ERRORS

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

EACCES

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

EFAULT

path points outside your accessible address space.

EIO

An I/O error occurred.

ELOOP

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

ENAMETOOLONG

path is too long.

ENOENT

The file does not exist.

ENOMEM

Insufficient kernel memory was available.

ENOTDIR

A component of *path* is not a directory.

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

EACCES

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

EBADF

fd is not a valid file descriptor.

SEE ALSO

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

NAME

clearerr, feof, ferror, fileno – check and reset stream status

SYNOPSIS

#include <stdio.h>

void clearerr(FILE * *stream*);
int feof(FILE * *stream*);
int ferror(FILE * *stream*);
int fileno(FILE * *stream*);

DESCRIPTION

The function **clearerr**() clears the end-of-file and error indicators for the stream pointed to by *stream*.

The function **feof**() tests the end-of-file indicator for the stream pointed to by *stream*, returning non-zero if it is set. The end-of-file indicator can only be cleared by the function **clearerr**().

The function **ferror**() tests the error indicator for the stream pointed to by *stream*, returning non-zero if it is set. The error indicator can only be reset by the **clearerr**() function.

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

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

ERRORS

These functions should not fail and do not set the external variable *errno*. (However, in case **fileno**() detects that its argument is not a valid stream, it must return −1 and set *errno* to **EBADF**.)

CONFORMING TO

The functions **clearerr**(), **feof**(), and **ferror**() conform to C89 and C99.

SEE ALSO

open(2), **fdopen**(3), **stdio**(3), **unlocked_stdio**(3)

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.

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 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_SEQ-PACKET**.

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

opendir / readdir – 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.

RETURN VALUE

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 */
};
```

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

printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf – formatted output conversion

SYNOPSIS

`#include <stdio.h>`

int printf(const char * format, ...);

int fprintf(FILE * stream, const char * format, ...);

int sprintf(char * str, const char * format, ...);

int snprintf(char * str, size_t size, const char * format, ...);

...

DESCRIPTION

The functions in the **printf()** family produce output according to a *format* as described below. The function **printf()** writes output to *stdout*, the standard output stream; **fprintf()** writes output to the given output *stream*; **sprintf()** and **snprintf()**, write to the character string *str*.

The function **snprintf()** writes at most *size* bytes (including the trailing null byte ('\0')) to *str*.

These functions write the output under the control of a *format* string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of **stdarg**(3)) are converted for output.

Return value

Upon successful return, these functions return the number of characters printed (not including the trailing '\0' used to end output to strings).

The functions **snprintf()** and **vsnprintf()** do not write more than *size* bytes (including the trailing '\0'). If the output was truncated due to this limit then the return value is the number of characters (not including the trailing '\0') which would have been written to the final string if enough space had been available. Thus, a return value of *size* or more means that the output was truncated.

If an output error is encountered, a negative value is returned.

Format of the format string

The format string is a character string, beginning and ending in its initial shift state, if any. The format string is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character **%**, and ends with a *conversion specifier*. In between there may be (in this order) zero or more *flags*, an optional minimum *field width*, an optional *precision* and an optional *length modifier*.

The conversion specifier

A character that specifies the type of conversion to be applied. An example for a conversion specifier is:

o, u, x, X

The *unsigned int* argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal (**x** and **X**) notation.

s

The *const char ** argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating null byte ('\0'); if a precision is specified, no more than the number specified are written. If a precision is given, no null byte need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating null byte.

SEE ALSO

print(1), **asprintf**(3), **dprintf**(3), **scanf**(3), **setlocale**(3), **wcrtomb**(3), **wprintf**(3), **locale**(5)

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

rename – change the name or location of a file

SYNOPSIS

#include <stdio.h>

int rename(const char *oldpath, const char *newpath);

DESCRIPTION

rename() renames a file, moving it between directories if required. Any other hard links to the file (as created using **link**(2)) are unaffected. Open file descriptors for *oldpath* are also unaffected.

If *newpath* already exists, it will be atomically replaced, so that there is no point at which another process attempting to access *newpath* will find it missing.

If *oldpath* and *newpath* are existing hard links referring to the same file, then **rename**() does nothing, and returns a success status.

If *newpath* exists but the operation fails for some reason, **rename**() guarantees to leave an instance of *newpath* in place.

RETURN VALUE

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

ERRORS

EACCES

Write permission is denied for the directory containing *oldpath* or *newpath*, or, search permission is denied for one of the directories in the path prefix of *oldpath* or *newpath*, or *oldpath* is a directory and does not allow write permission (needed to update the .. entry). (See also **path_resolution**(7))

EINVAL

The new pathname contained a path prefix of the old, or, more generally, an attempt was made to make a directory a subdirectory of itself.

EISDIR

newpath is an existing directory, but *oldpath* is not a directory.

ENAMETOOLONG

oldpath or *newpath* was too long.

ENOENT

The link named by *oldpath* does not exist, or, a directory component in *newpath* does not exist, or, *oldpath* or *newpath* is an empty string.

ENOMEM

Insufficient kernel memory was available.

CONFORMING TO

rename(): 4.3BSD, C89, C99, POSIX.1-2001, POSIX.1-2008.

SEE ALSO

mv(1), **chmod**(2), **link**(2), **symlink**(2), **unlink**(2), **path_resolution**(7), **symlink**(7)

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 call to **strtok**() the string to be parsed should be specified in *str*. In each subsequent call that should parse the same string, *str* must be NULL.

The *delim* argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may 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 does 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 point from which to start searching for the next token. The first call to **strtok**() sets this pointer to point to the first byte of the string. The start of the next token is determined by scanning forward for the next nondelimiter byte in *str*. If such a byte is found, it is taken as the start of the next token. If no such byte is found, then there are no more tokens, and **strtok**() returns NULL. (A string that is empty or that contains only 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 the terminating null byte ('\0') is encountered. If a delimiter byte is found, it is overwritten with a null byte to terminate the current token, and **strtok**() saves a pointer to the following byte; that pointer will be used as the starting point when searching for the next token. In this case, **strtok**() returns a pointer to the start of the found token.

From the above description, it follows that a sequence of two or more contiguous delimiter bytes in the parsed string is considered to be a single delimiter, and that delimiter bytes at the start or end of the string are ignored. Put another way: the tokens returned by **strtok**() are always nonempty strings. Thus, for 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 ** variable that is used internally by **strtok_r**() in order to maintain context between successive calls that parse the same string. On the first call to **strtok_r**(), *str* should point to the string to be parsed, and the value of *saveptr* is ignored. In subsequent calls, *str* should be NULL, and *saveptr* should be unchanged since the previous call.

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

RETURN VALUE

The **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.