NAME

accept — accept a new connection on a socket

SYNOPSIS

#include <sys/socket.h>

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

RETURN VALUE

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

ERRORS

The *accept*() function shall fail if:

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

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

ECONNABORTED
A connection has been aborted.

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

EINVAL
The *socket* is not accepting connections.

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

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

ENOBUFS
No buffer space is available.

ENOMEM
There was insufficient memory available to complete the operation.

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

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

The *accept*() function may fail if:

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

APPLICATION USAGE

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

NAME

bind — bind a name to a socket

SYNOPSIS

#include <sys/socket.h>

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

RETURN VALUE

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

ERRORS

The *bind*() function shall fail if:

EADDRINUSE
The specified address is already in use.

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

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

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

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

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

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

NAME

chdir — change working directory

SYNOPSIS

#include <unistd.h>

int chdir(const char **path*);

DESCRIPTION

The *chdir*() function shall cause the directory named by the pathname pointed to by the *path* argument to become the current working directory; that is, the starting point for path searches for pathnames not beginning with **'/'**.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned, the current working directory shall remain unchanged, and *errno* shall be set to indicate the error.

ERRORS

The *chdir*() function shall fail if:

EACCES
Search permission is denied for any component of the pathname.

ELOOP
A loop exists in symbolic links encountered during resolution of the *path* argument.

ENAMETOOLONG
The length of a component of a pathname is longer than {NAME_MAX}.

ENOENT
A component of *path* does not name an existing directory or *path* is an empty string.

ENOTDIR
A component of the pathname names an existing file that is neither a directory nor a symbolic link to a directory.

The *chdir*() function may fail if:

ELOOP
More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

ENAMETOOLONG
The length of a pathname exceeds {PATH_MAX}, or pathname resolution of a symbolic link produced an intermediate result with a length that exceeds {PATH_MAX}.

The following sections are informative.

EXAMPLES

Changing the Current Working Directory

The following example makes the value pointed to by **directory**, **/tmp**, the current working directory.

```
#include <unistd.h>
...
char *directory = "/tmp";
int ret;

ret = chdir (directory);
```

RATIONALE

The *chdir*() function only affects the working directory of the current process.

NAME

opendir — open directory

SYNOPSIS

#include <dirent.h>

DIR *opendir(const char *_dirname_);

DESCRIPTION

The *opendir*() function shall open a directory stream corresponding to the directory named by the *dirname* argument. The directory stream is positioned at the first entry. If the type **DIR** is implemented using a file descriptor, applications shall only be able to open up to a total of {OPEN_MAX} files and directories.

If the type **DIR** is implemented using a file descriptor, the descriptor shall be obtained as if the O_DIRECTORY flag was passed to *open*().

RETURN VALUE

Upon successful completion, the function shall return a pointer to an object of type **DIR**. Otherwise, the function shall return a null pointer and set *errno* to indicate the error.

ERRORS

The *opendir*() function shall fail if:

EACCES

Search permission is denied for the component of the path prefix of *dirname* or read permission is denied for *dirname*.

ELOOP

A loop exists in symbolic links encountered during resolution of the *dirname* argument.

ENAMETOOLONG

The length of a component of a pathname is longer than {NAME_MAX}.

ENOENT

A component of *dirname* does not name an existing directory or *dirname* is an empty string.

ENOTDIR

A component of *dirname* names an existing file that is neither a directory nor a symbolic link to a directory.

NAME

fprintf, printf, snprintf, sprintf — print formatted output

SYNOPSIS

#include <stdio.h>

int fprintf(FILE *restrict *stream*, const char *restrict *format*, ...);
int printf(const char *restrict *format*, ...);
int snprintf(char *restrict *s*, size_t *n*,
const char *restrict *format*, ...);
int sprintf(char *restrict *s*, const char *restrict *format*, ...);

DESCRIPTION

The *fprintf*() function shall place output on the named output *stream*. The *printf*() function shall place output on the standard output stream *stdout*. The *sprintf*() function shall place output followed by the null byte, **'\0'**, in consecutive bytes starting at *_s_; it is the user's responsibility to ensure that enough space is available.

The *snprintf*() function shall be equivalent to *sprintf*(), with the addition of the *n* argument which states the size of the buffer referred to by *s*. If *n* is zero, nothing shall be written and *s* may be a null pointer. Otherwise, output bytes beyond the *n*-1st shall be discarded instead of being written to the array, and a null byte is written at the end of the bytes actually written into the array.

If copying takes place between objects that overlap as a result of a call to *sprintf*() or *snprintf*(), the results are undefined.

Each of these functions converts, formats, and prints its arguments under control of the *format*. The *format* is a character string, beginning and ending in its initial shift state, if any. The *format* is composed of zero or more directives: *ordinary characters*, which are simply copied to the output stream, and *conversion specifications*, each of which shall result in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the *format*. If the *format* is exhausted while arguments remain, the excess arguments shall be evaluated but are otherwise ignored.

Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than to the next unused argument. In this case, the conversion specifier character **%** (see below) is replaced by the sequence "%*n*$", where *n* is a decimal integer in the range [1,{NL_ARGMAX}], giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages (see the EXAMPLES section).

The *format* can contain either numbered argument conversion specifications (that is, "%*n*$" and "*_m_$"), or unnumbered argument conversion specifications (that is, **%** and *), but not both. The only exception to this is that **%%** can be mixed with the "%*n*$" form. The results of mixing numbered and unnumbered argument specifications in a *format* string are undefined. When numbered argument specifications are used, specifying the *N*th argument requires that all the leading arguments, from the first to the (*N*−*1*)th, are specified in the format string.

In format strings containing the **%** form of conversion specification, each conversion specification uses the first unused argument in the argument list.

Each conversion specification is introduced by the **'%'** character or by the character sequence "%*n*$", after which the following appear in sequence:

* Zero or more *flags* (in any order), which modify the meaning of the conversion specification.
* An optional minimum *field width*.
* An optional *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, and **X** conversion specifiers
* An optional length modifier that specifies the size of the argument.
* A *conversion specifier* character that indicates the type of conversion to be applied.

The conversion specifiers and their meanings are:

d, i
: The **int** argument shall be converted to a signed decimal in the style "[−]*dddd*". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

c
: The **int** argument shall be converted to an **unsigned char**, and the resulting byte shall be written.

s
: The argument shall be a pointer to an array of **char**. Bytes from the array shall be written up to (but not including) any terminating null byte. If the precision is specified, no more than that many bytes shall be written. If the precision is not specified or is greater than the size of the array, the application shall ensure that the array contains a null byte.

p
: The argument shall be a pointer to **void**. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner.

If a conversion specification does not match one of the above forms, the behavior is undefined. If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

In no case shall a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field shall be expanded to contain the conversion result. Characters generated by *fprintf*() and *printf*() are printed as if *fputc*() had been called.

The last data modification and last file status change timestamps of the file shall be marked for update between the call to a successful execution of *fprintf*() or *printf*() and the next successful completion of a call to *fflush*() or *fclose*() on the same stream or a call to *exit*() or *abort*()

RETURN VALUE

Upon successful completion, the *dprintf*(), *fprintf*(), and *printf*() functions shall return the number of bytes transmitted.

Upon successful completion, the *sprintf*() function shall return the number of bytes written to *s*, excluding the terminating null byte.

Upon successful completion, the *snprintf*() function shall return the number of bytes that would be written to *s* had *n* been sufficiently large excluding the terminating null byte.

If an output error was encountered, these functions shall return a negative value and set *errno* to indicate the error.

If the value of *n* is zero on a call to *snprintf*(), nothing shall be written, the number of bytes that would have been written had *n* been sufficiently large excluding the terminating null shall be returned, and *s* may be a null pointer.

ERRORS

For the conditions under which *fprintf*(), and *printf*() fail and may fail, refer to *fputc*( ) or *fputwc*( ).

In addition, all forms of *fprintf*() shall fail if:

EILSEQ
: A wide-character code that does not correspond to a valid character has been detected.

EOVERFLOW
: The value to be returned is greater than {INT_MAX}.

The *fprintf*(), and *printf*() functions may fail if:

ENOMEM
: Insufficient storage space is available.

The *snprintf*() function shall fail if:

EOVERFLOW
: The value of *n* is greater than {INT_MAX}.

NAME

fstatat, lstat, stat — get file status

SYNOPSIS

```
#include <fcntl.h>
#include <sys/stat.h>

int fstatat(int fd, const char *restrict path,
    struct stat *restrict buf, int flag);
int lstat(const char *restrict path, struct stat *restrict buf);
int stat(const char *restrict path, struct stat *restrict buf);
```

DESCRIPTION

The *stat*() function shall obtain information about the named file and write it to the area pointed to by the *buf* argument. The *path* argument points to a pathname naming a file. Read, write, or execute permission of the named file is not required. An implementation that provides additional or alternate file access control mechanisms may, under implementation-defined conditions, cause *stat*() to fail. In particular, the system may deny the existence of the file specified by *path*.

If the named file is a symbolic link, the *stat*() function shall continue pathname resolution using the contents of the symbolic link, and shall return information pertaining to the resulting file if the file exists.

The *buf* argument is a pointer to a **stat** structure, as defined in the *<sys/stat.h>* header, into which information is placed concerning the file.

The *stat*() function shall update any time-related fields (as described in the Base Definitions volume of POSIX.1-2017, *Section 4.9*, *File Times Update*), before writing into the **stat** structure.

If the named file is a shared memory object, the implementation shall update in the **stat** structure pointed to by the *buf* argument the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be valid. The implementation may update other fields and flags.

If the named file is a typed memory object, the implementation shall update in the **stat** structure pointed to by the *buf* argument the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be valid. The implementation may update other fields and flags.

For all other file types defined in this volume of POSIX.1-2017, the structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atim*, *st_ctim*, and *st_mtim* shall have meaningful values and the value of the member *st_nlink* shall be set to the number of links to the file.

The *lstat*() function shall be equivalent to *stat*(), except when *path* refers to a symbolic link. In that case *lstat*() shall return information about the link, while *stat*() shall return information about the file the link references.

For symbolic links, the *st_mode* member shall contain meaningful information when used with the file type macros. The file mode bits in *st_mode* are unspecified. The structure members *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atim*, *st_ctim*, and *st_mtim* shall have meaningful values and the value of the *st_nlink* member shall be set to the number of (hard) links to the symbolic link. The value of the *st_size* member shall be set to the length of the pathname contained in the symbolic link not including any terminating null byte.

The *fstatat*() function shall be equivalent to the *stat*() or *lstat*() function, depending on the value of *flag* (see below), except in the case where *path* specifies a relative path. In this case the status shall be retrieved from a file relative to the directory associated with the file descriptor *fd* instead of the current working directory. If the access mode of the open file description associated with the file descriptor is not O_SEARCH, the function shall check whether directory searches are permitted using the current permissions of the directory underlying the file descriptor. If the access mode is O_SEARCH, the function shall not perform the check.

Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in *<fcntl.h>*:

AT_SYMLINK_NOFOLLOW
If *path* names a symbolic link, the status of the symbolic link is returned.

If *fstatat*() is passed the special value AT_FDCWD in the *fd* parameter, the current working directory shall be used and the behavior shall be identical to a call to *stat*() or *lstat*() respectively, depending on whether or not the AT_SYMLINK_NOFOLLOW bit is set in *flag*.

RETURN VALUE

Upon successful completion, these functions shall return 0. Otherwise, these functions shall return −1 and set *errno* to indicate the error.

ERRORS

These functions shall fail if:

EACCES
Search permission is denied for a component of the path prefix.

EIO An error occurred while reading from the file system.

ELOOP
A loop exists in symbolic links encountered during resolution of the *path* argument.

ENAMETOOLONG
The length of a component of a pathname is longer than {NAME_MAX}.

ENOENT
A component of *path* does not name an existing file or *path* is an empty string.

ENOTDIR
A component of the path prefix names an existing file that is neither a directory nor a symbolic link to a directory, or the *path* argument contains at least one non-<slash> character and ends with one or more trailing <slash> characters and the last pathname component names an existing file that is neither a directory nor a symbolic link to a directory.

EOVERFLOW
The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by *buf*.

EXAMPLES

Obtaining File Status Information

The following example shows how to obtain file status information for a file named **/home/cnd/mod1**. The structure variable *buffer* is defined for the **stat** structure. Error handling is ommitted for brevity.

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

struct stat buffer;
int         status;
...
status = stat("/home/cnd/mod1", &buffer);
```


NAME

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

SYNOPSIS

#include <sys/socket.h>

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

DESCRIPTION

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

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

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

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

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

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

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

RETURN VALUE

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

ERRORS

The *listen*() function shall fail if:

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

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

EINVAL
The *socket* is already connected.

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

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

The *listen*() function may fail if:

EACCES
The calling process does not have appropriate privileges.

EINVAL
The *socket* has been shut down.

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

NAME

pthread_create — thread creation

SYNOPSIS

#include <pthread.h>

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

RETURN VALUE

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

ERRORS

The *pthread_create*() function shall fail if:

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

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

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

NAME

pthread_detach — detach a thread

SYNOPSIS

#include <pthread.h>

int pthread_detach(pthread_t *thread*);

DESCRIPTION

The *pthread_detach*() function shall indicate to the implementation that storage for the thread *thread* can be reclaimed when that thread terminates. If *thread* has not terminated, *pthread_detach*() shall not cause it to terminate.

The behavior is undefined if the value specified by the *thread* argument to *pthread_detach*() does not refer to a joinable thread.

RETURN VALUE

If the call succeeds, *pthread_detach*() shall return 0; otherwise, an error number shall be returned to indicate the error.

ERRORS

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

The following sections are informative.

RATIONALE

The *pthread_join*() or *pthread_detach*() functions should eventually be called for every thread that is created so that storage associated with the thread may be reclaimed.

It has been suggested that a ‘‘detach’’ function is not necessary; the *detachstate* thread creation attribute is sufficient, since a thread need never be dynamically detached. However, need arises in at least two cases:

1. In a cancellation handler for a *pthread_join*() it is nearly essential to have a *pthread_detach*() function in order to detach the thread on which *pthread_join*() was waiting. Without it, it would be necessary to have the handler do another *pthread_join*() to attempt to detach the thread, which would both delay the cancellation processing for an unbounded period and introduce a new call to *pthread_join*(), which might itself need a cancellation handler. A dynamic detach is nearly essential in this case.
2. In order to detach the ‘‘initial thread’’ (as may be desirable in processes that set up server threads).

If an implementation detects that the value specified by the *thread* argument to *pthread_detach*() does not refer to a joinable thread, it is recommended that the function should fail and report an **[EINVAL]** error.

If an implementation detects use of a thread ID after the end of its lifetime, it is recommended that the function should fail and report an **[ESRCH]** error.

NAME

readdir, readdir_r — read a directory

SYNOPSIS

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
int readdir_r(DIR *restrict dirp, struct dirent *restrict entry,
    struct dirent **restrict result);
```

DESCRIPTION

The type **DIR**, which is defined in the *<dirent.h>* header, represents a *directory stream*, which is an ordered sequence of all the directory entries in a particular directory. Directory entries represent files; files may be removed from a directory or added to a directory asynchronously to the operation of *readdir*().

The *readdir*() function shall return a pointer to a structure representing the directory entry at the current position in the directory stream specified by the argument *dirp*, and position the directory stream at the next entry. It shall return a null pointer upon reaching the end of the directory stream. The structure **dirent** defined in the *<dirent.h>* header describes a directory entry. The value of the structure's *d_ino* member shall be set to the file serial number of the file named by the *d_name* member. If the *d_name* member names a symbolic link, the value of the *d_ino* member shall be set to the file serial number of the symbolic link itself.

The *readdir*() function shall not return directory entries containing empty names. If entries for dot or dot-dot exist, one entry shall be returned for dot and one entry shall be returned for dot-dot; otherwise, they shall not be returned.

The application shall not modify the structure to which the return value of *readdir*() points, nor any storage areas pointed to by pointers within the structure. The returned pointer, and pointers within the structure, might be invalidated or the structure or the storage areas might be overwritten by a subsequent call to *readdir*() on the same directory stream. They shall not be affected by a call to *readdir*() on a different directory stream. The returned pointer, and pointers within the structure, might also be invalidated if the calling thread is terminated.

The *readdir*() function may buffer several directory entries per actual read operation; *readdir*() shall mark for update the last data access timestamp of the directory each time the directory is actually read.

After a call to *fork*(), either the parent or child (but not both) may continue processing the directory stream using *readdir*(), *rewinddir*(), or *seekdir*(). If both the parent and child processes use these functions, the result is undefined.

The *readdir*() function need not be thread-safe.

Applications wishing to check for error situations should set *errno* to 0 before calling *readdir*(). If *errno* is set to non-zero on return, an error occurred.

The *readdir_r*() function shall initialize the **dirent** structure referenced by *entry* to represent the directory entry at the current position in the directory stream referred to by *dirp*, store a pointer to this structure at the location referenced by *result*, and position the directory stream at the next entry.

The storage pointed to by *entry* shall be large enough for a **dirent** with an array of **char** *d_name* members containing at least {NAME_MAX}+1 elements.

Upon successful return, the pointer returned at **result* shall have the same value as the argument *entry*. Upon reaching the end of the directory stream, this pointer shall have the value NULL.

The *readdir_r*() function shall not return directory entries containing empty names.

The *readdir_r*() function may buffer several directory entries per actual read operation; *readdir_r*() shall mark for update the last data access timestamp of the directory each time the directory is actually read.

RETURN VALUE

Upon successful completion, *readdir*() shall return a pointer to an object of type **struct dirent**. When an error is encountered, a null pointer shall be returned and *errno* shall be set to indicate the error. When the

end of the directory is encountered, a null pointer shall be returned and *errno* is not changed.

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

ERRORS

These functions shall fail if:

EOVERFLOW
One of the values in the structure to be returned cannot be represented correctly.

These functions may fail if:

EBADF
The *dirp* argument does not refer to an open directory stream.

ENOENT
The current position of the directory stream is invalid.

EXAMPLES

The following sample program searches the current directory for each of the arguments supplied on the command line. Some error handling code is omitted for brevity.

```
#include <dirent.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>

static void lookup(const char *arg) {
   DIR *dirp;
   struct dirent *dp;
   if ((dirp = opendir(".")) == NULL) {
      perror("couldn't open '.'");
      return;
   }
   do {
      errno = 0;
      if ((dp = readdir(dirp)) != NULL) {
         if (strcmp(dp->d_name, arg) != 0) continue;
         (void) printf("found %s\n", arg);
         (void) closedir(dirp);
         return;
      }
   } while (dp != NULL);
   if (errno != 0) perror("error reading directory");
   else (void) printf("failed to find %s\n", arg);
   (void) closedir(dirp);
   return;
}

int main(int argc, char *argv[]) {
   for (int i = 1; i < argc; i++) lookup(argv[i]);
   return (0);
}
```

NAME

socket — create an endpoint for communication

SYNOPSIS

#include <sys/socket.h>

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

DESCRIPTION

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

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

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

type Specifies the type of socket to be created.

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

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

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

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

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

SOCK_DGRAM

SOCK_SEQPACKET

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

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

RETURN VALUE

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

ERRORS

The *socket*() function shall fail if:

EAFNOSUPPORT
The implementation does not support the specified address family.

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

ENFILE
No more file descriptors are available for the system.

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

EPROTOTYPE
The socket type is not supported by the protocol.

NAME

strtok, strtok_r — split string into tokens

SYNOPSIS

#include <string.h>

char *strtok(char *restrict *s*, const char *restrict *sep*);
char *strtok_r(char *restrict *s*, const char *restrict *sep*,
char **restrict *state*);

DESCRIPTION

For *strtok*(): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2017 defers to the ISO C standard.

A sequence of calls to *strtok*() breaks the string pointed to by *s* into a sequence of tokens, each of which is delimited by a byte from the string pointed to by *sep*. The first call in the sequence has *s* as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by *sep* may be different from call to call.

The first call in the sequence searches the string pointed to by *s* for the first byte that is *not* contained in the current separator string pointed to by *sep*. If no such byte is found, then there are no tokens in the string pointed to by *s* and *strtok*() shall return a null pointer. If such a byte is found, it is the start of the first token.

The *strtok*() function then searches from there for a byte that *is* contained in the current separator string. If no such byte is found, the current token extends to the end of the string pointed to by *s*, and subsequent searches for a token shall return a null pointer. If such a byte is found, it is overwritten by a NUL character, which terminates the current token. The *strtok*() function saves a pointer to the following byte, from which the next search for a token shall start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

The implementation shall behave as if no function defined in this volume of POSIX.1-2017 calls *strtok*().

The *strtok*() function need not be thread-safe.

The *strtok_r*() function shall be equivalent to *strtok*(), except that *strtok_r*() shall be thread-safe and the argument *state* points to a user-provided pointer that allows *strtok_r*() to maintain state between calls which scan the same string. The application shall ensure that the pointer pointed to by *state* is unique for each string (*s*) being processed concurrently by *strtok_r*() calls. The application need not initialize the pointer pointed to by *state* to any particular value. The implementation shall not update the pointer pointed to by *state* to point (directly or indirectly) to resources, other than within the string *s*, that need to be freed or released by the caller.

RETURN VALUE

Upon successful completion, *strtok*() shall return a pointer to the first byte of a token. Otherwise, if there is no token, *strtok*() shall return a null pointer.

The *strtok_r*() function shall return a pointer to the token found, or a null pointer when no token is found.

ERRORS

No errors are defined.