NAME

opendir − open a directory / readdir − read a directory

SYNOPSIS

#include <sys/types.h>

#include <dirent.h>

DIR *opendir(const char ****name*);**

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

DESCRIPTION opendir

The **opendir()** function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

RETURN VALUE

The **opendir()** function returns a pointer to the directory stream or NULL if an error occurred.

DESCRIPTION readdir

The **readdir()** function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by *dir*. It returns NULL on reaching the end-of-file or if an error occurred.

DESCRIPTION readdir_r

The **readdir_r()** function initializes the structure referenced by *entry* and stores a pointer to this structure in *result*. On successful return, the pointer returned at **result* will have the same value as the argument *entry*. Upon reaching the end of the directory stream, this pointer will have the value NULL.

The data returned by **readdir()** is overwritten by subsequent calls to **readdir()** for the **same** directory stream.

The *dirent* structure is defined as follows:

```
struct dirent {
    long          d_ino;          /* inode number */
    off_t         d_off;          /* offset to the next dirent */
    unsigned short  d_reclen;     /* length of this record */
    unsigned char   d_type;     /* type of file */
    char          d_name[256];  /* filename */
};
```

RETURN VALUE

The **readdir()** function returns a pointer to a dirent structure, or NULL if an error occurs or end-of-file is reached.

readdir_r() returns 0 if successful or an error number to indicate failure.

ERRORS

EACCES
Permission denied.

ENOENT
Directory does not exist, or *name* is an empty string.

ENOTDIR
name is not a directory.

NAME

fnmatch − match filename or pathname

SYNOPSIS

#include <fnmatch.h>

int fnmatch(const char ****pattern*, const char ****string***, int** *flags***);**

DESCRIPTION

The **fnmatch**() function checks whether the *string* argument matches the *pattern* argument, which is a shell wildcard pattern.

The *flags* argument modifies the behavior; it is the bitwise OR of zero or more of the following flags:

FNM_NOESCAPE
If this flag is set, treat backslash as an ordinary character, instead of an escape character.

FNM_PATHNAME
If this flag is set, match a slash in *string* only with a slash in *pattern* and not by an asterisk (*) or a question mark (?) metacharacter, nor by a bracket expression ([]) containing a slash.

FNM_PERIOD
If this flag is set, a leading period in *string* has to be matched exactly by a period in *pattern*. A period is considered to be leading if it is the first character in *string*, or if both **FNM_PATHNAME** is set and the period immediately follows a slash.

FNM_FILE_NAME
This is a GNU synonym for **FNM_PATHNAME**.

FNM_LEADING_DIR
If this flag (a GNU extension) is set, the pattern is considered to be matched if it matches an initial segment of *string* which is followed by a slash. This flag is mainly for the internal use of glibc and is only implemented in certain cases.

FNM_CASEFOLD
If this flag (a GNU extension) is set, the pattern is matched case-insensitively.

RETURN VALUE

Zero if *string* matches *pattern*, **FNM_NOMATCH** if there is no match or another nonzero value if there is an error.

CONFORMING TO

POSIX.2. The **FNM_FILE_NAME**, **FNM_LEADING_DIR**, and **FNM_CASEFOLD** flags are GNU extensions.

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

gets, fgets − get a string from a stream
fputs, puts − output of strings

SYNOPSIS

#include <stdio.h>

char *gets(char ****s*);**

char *fgets(char ****s*, int** *n***, FILE *****stream***);**

int fputs(const char ****s*, FILE *****stream***);**

int puts(const char ****s*);**

DESCRIPTION gets/fgets

The **gets( )** function reads characters from the standard input stream (see **intro**(3)), **stdin**, into the array pointed to by *s*, until a newline character is read or an end-of-file condition is encountered. The newline character is discarded and the string is terminated with a null character.

The **fgets( )** function reads characters from the *stream* into the array pointed to by *s*, until *n*−1 characters are read, or a newline character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

When using **gets( )**, if the length of an input line exceeds the size of *s*, indeterminate behavior may result. For this reason, it is strongly recommended that **gets( )** be avoided in favor of **fgets( )**.

RETURN VALUES

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a null pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a null pointer is returned and the error indicator for the stream is set. If end-of-file is encountered, the **EOF** indicator for the stream is set. Otherwise *s* is returned.

ERRORS

The **gets( )** and **fgets( )** functions will fail if data needs to be read and:

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

DESCRIPTION puts/fputs

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

puts() writes the string *s* and a trailing newline to *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

puts() and **fputs()** return a non - negative number on success, or **EOF** on error.

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

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 functions **printf**() and **vprintf**() write output to *stdout*, the standard output stream; **fprintf**() and **vfprintf**() write output to the given output *stream*; **sprintf**(), **snprintf**(), **vsprintf**() and **vsnprintf**() write to the character string *str*.

The functions **snprintf**() and **vsnprintf**() write at most *size* bytes (including the trailing null byte ('\0')) to *str*.

The functions **vprintf**(), **vfprintf**(), **vsprintf**(), **vsnprintf**() are equivalent to the functions **printf**(), **fprintf**(), **sprintf**(), **snprintf**(), respectively, except that they are called with a *va_list* instead of a variable number of arguments. These functions do not call the *va_end* macro. Because they invoke the *va_arg* macro, the value of *ap* is undefined after the call. See **stdarg**(3).

These eight 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. (See also below under NOTES.)

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 arguments must correspond properly (after type promotion) with the conversion specifier. By default, the arguments are used in the order given, where each '*' and each conversion specifier asks for the next argument (and it is an error if insufficiently many arguments are given). One can also specify explicitly which argument is taken, at each place where an argument is required, by writing "%m$" instead of '%' and "*m$" instead of '*', where the decimal integer m denotes the position in the argument list of the desired argument, indexed starting from 1. Thus,

```
printf("%*d", width, num);
```


and

```
printf("%2$*1$d", width, num);
```

are equivalent. The second style allows repeated references to the same argument. The C99 standard does not include the style using '$', which comes from the Single Unix Specification. If the style using '$' is used, it must be used throughout for all conversions taking an argument and all width and precision arguments, but it may be mixed with "%%" formats which do not consume an argument. There may be no gaps in the numbers of arguments specified using '$'; for example, if arguments 1 and 3 are specified, argument 2 must also be specified somewhere in the format string.

For some numeric conversions a radix character ("decimal point") or thousands' grouping character is used. The actual character used depends on the **LC_NUMERIC** part of the locale. The POSIX locale uses '.' as radix character, and does not have a grouping character. Thus,

```
printf("%'.2f", 1234567.89);
```

results in "1234567.89" in the POSIX locale, in "1234567,89" in the nl_NL locale, and in "1.234.567,89" in the da_DK locale.

The conversion specifier

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

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

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

COLOPHON

This page is part of release 3.05 of the Linux *man-pages* project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.

NAME

stat, fstat, lstat − get file status

SYNOPSIS

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

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

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

lstat(): _BSD_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION

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

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

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

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

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

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

The *st_dev* field describes the device on which this file resides.

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

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

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

The *st_blksize* field gives the "preferred" blocksize for efficient file system I/O. (Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.)

Not all of the Linux file systems implement all of the time fields. Some file system types allow mounting in such a way that file accesses do not cause an update of the *st_atime* field. (See "noatime" in **mount**(8).)

The field *st_atime* is changed by file accesses, for example, by **execve**(2), **mknod**(2), **pipe**(2), **utime**(2) and **read**(2) (of more than zero bytes). Other routines, like **mmap**(2), may or may not update *st_atime*.

The field *st_mtime* is changed by file modifications, for example, by **mknod**(2), **truncate**(2), **utime**(2) and **write**(2) (of more than zero bytes). Moreover, *st_mtime* of a directory is changed by the creation or deletion of files in that directory. The *st_mtime* field is *not* changed for changes in owner, group, hard link count, or mode.

The field *st_ctime* is changed by writing or by setting inode information (i.e., owner, group, link count, mode, etc.).

The following POSIX macros are defined to check the file type using the *st_mode* field:

S_ISREG(m)	is it a regular file?
S_ISDIR(m)	directory?
S_ISCHR(m)	character device?
S_ISBLK(m)	block device?
S_ISFIFO(m)	FIFO (named pipe)?
S_ISLNK(m)	symbolic link? (Not in POSIX.1-1996.)
S_ISSOCK(m)	socket? (Not in POSIX.1-1996.)

RETURN VALUE

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

ERRORS

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

EBADF
fd is bad.

EFAULT
Bad address.

ELOOP
Too many symbolic links encountered while traversing the path.

ENAMETOOLONG
File name too long.

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

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

ENOTDIR
A component of the path is not a directory.

SEE ALSO

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