

fopen/fdopen/fileno(3)

fopen/fdopen/fileno(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)**

getc/fgets/putc/fputs(3)

getc/fgets/putc/fputs(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)**, **fgetc(3)**, **fgetwc(3)**, **fgetws(3)**, **fopen(3)**, **fread(3)**, **fseek(3)**, **getline(3)**, **getwchar(3)**, **scanf(3)**, **ungetc(3)**, **write(2)**, **ferror(3)**, **fopen(3)**, **fputc(3)**, **fputwc(3)**, **fputws(3)**, **fseek(3)**, **fwrite(3)**, **gets(3)**, **putwchar(3)**, **scanf(3)**, **unlocked_stdio(3)**

opendir/readdir(3)

opendir/readdir(3)

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);
```

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. It is safe to use **readdir()** inside threads if the pointers passed as *dir* are created by distinct calls to **opendir()**.

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; not supported by all filesystem types */
    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.

ERRORS

EACCES

Permission denied.

ENOENT

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

ENOTDIR

name is not a directory.

pthread_create/pthread_exit(3)

pthread_create/pthread_exit(3)

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)**.

pthread_detach(pthread_self(3))

pthread_detach(pthread_self(3))

NAME

pthread_detach – put a running thread in the detached state

pthread_self – obtain ID of the calling thread

SYNOPSIS

```
#include <pthread.h>
```

```
int pthread_detach(pthread_t th);
```

```
pthread_t pthread_self(void);
```

DESCRIPTION pthread_detach

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.

DESCRIPTION pthread_self

function returns the ID of the calling thread. This is the same value that is returned in **thread* in the **pthread_create(3)** call that created this thread.

RETURN VALUE

The **pthread_detach()** function returns 0 on success. On error, a non-zero error code is returned.

The **pthread_self()** function always succeeds, returning the calling thread's ID.

ERRORS

ESRCH

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

EINVAL

the thread *th* is already in the detached state

stat(2)

stat(2)

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;  /* blocksizes 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" blocksizes for efficient file system I/O. (Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.)

stat(2)

stat(2)

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)**

strdup(3)

strdup(3)

NAME

strdup, **strndup** – duplicate a string

SYNOPSIS

```
#include <string.h>
char *strdup(const char *s);
char *strndup(const char *s, size_t n);
```

DESCRIPTION

The **strdup()** function returns a pointer to a new string which is a duplicate of the string *s*. Memory for the new string is obtained with **malloc(3)**, and can be freed with **free(3)**.

The **strndup()** function is similar, but copies at most *n* bytes. If *s* is longer than *n*, only *n* bytes are copied, and a terminating null byte (`'\0'`) is added.

RETURN VALUE

On success, the **strdup()** function returns a pointer to the duplicated string. It returns NULL if insufficient memory was available, with *errno* set to indicate the cause of the error.

ERRORS

ENOMEM

Insufficient memory available to allocate duplicate string.

CONFORMING TO

strdup() conforms to SVr4, 4.3BSD, POSIX.1-2001. **strndup()** conforms to POSIX.1-2008.