NAME

fopen, fdopen, fileno – stream open functions

SYNOPSIS

#include <stdio.h>

FILE *fopen(const char *_path_, const char *_mode_);
FILE *fdopen(int _fildes_, const char *_mode_);
int fileno(FILE *_stream_);
int fclose(FILE *_stream_);

DESCRIPTION

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

The **fclose**() function flushes the stream pointed to by *stream* (writing any buffered output data using **fflush**(3)) and closes the underlying file 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. Upon successful completion of **fclose**, 0 is returned. Otherwise, **EOF** is returned and *errno* is set to indicate the error.

ERRORS

EINVAL

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

EBADF

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

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

NAME

getuid, geteuid – get user identity

SYNOPSIS

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

uid_t getuid(void);
uid_t geteuid(void);

DESCRIPTION

getuid() returns the real user ID of the calling process.

geteuid() returns the effective user ID of the calling process.

ERRORS

These functions are always successful.

CONFORMING TO

POSIX.1-2001, POSIX.1-2008, 4.3BSD.

NOTES

History

In UNIX V6 the **getuid**() call returned *(euid << 8) + uid*. UNIX V7 introduced separate calls **getuid**() and **geteuid**().

The original Linux **getuid**() and **geteuid**() system calls supported only 16-bit user IDs. Subsequently, Linux 2.4 added **getuid32**() and **geteuid32**(), supporting 32-bit IDs. The glibc **getuid**() and **geteuid**() wrapper functions transparently deal with the variations across kernel versions.

On Alpha, instead of a pair of **getuid**() and **geteuid**() system calls, a single **getxuid**() system call is provided, which returns a pair of real and effective UIDs. The glibc **getuid**() and **geteuid**() wrapper functions transparently deal with this. See **syscall**(2) for details regarding register mapping.

SEE ALSO

getresuid(2), **setreuid**(2), **setuid**(2), **credentials**(7)

COLOPHON

This page is part of release 5.10 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man−pages/.

NAME

opendir – open a directory / readdir – read a directory

SYNOPSIS

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

DIR *opendir(const char *_name_);
int closedir(DIR *_dirp_);
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. On error, NULL is returned, and *errno* is set appropriately.

DESCRIPTION closedir

The **closedir()** function closes the directory stream associated with *dirp*. A successful call to **closedir()** also closes the underlying file descriptor associated with *dirp*. The directory stream descriptor *dirpP is not available after this call*.

RETURN VALUE

The **closedir()** function returns 0 on success. On error, −1 is returned, and *errno* is set appropriately.

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 */
    char        d_name[256]; /* filename */
};
```

RETURN VALUE

On success, **readdir**() returns a pointer to a *dirent* structure. (This structure may be statically allocated; do not attempt to **free**(3) it.)

If the end of the directory stream is reached, NULL is returned and *errno* is not changed. If an error occurs, NULL is returned and *errno* is set appropriately. To distinguish end of stream and from an error, set *errno* to zero before calling **readdir**() and then check the value of *errno* if NULL is returned.

ERRORS

EACCES
Permission denied.

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

ENOTDIR
name is not a directory.

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

pthread_self – obtain ID of the calling thread

SYNOPSIS

#include <pthread.h>

pthread_t pthread_self(void);

Compile and link with *–pthread*.

DESCRIPTION

The **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

This function always succeeds, returning the calling thread's ID.

ERRORS

This function always succeeds.

NOTES

POSIX.1 allows an implementation wide freedom in choosing the type used to represent a thread ID; for example, representation using either an arithmetic type or a structure is permitted. Therefore, variables of type *pthread_t* can't portably be compared using the C equality operator (**==**); use **pthread_equal**(3) instead.

Thread identifiers should be considered opaque: any attempt to use a thread ID other than in pthreads calls is nonportable and can lead to unspecified results.

Thread IDs are guaranteed to be unique only within a process. A thread ID may be reused after a terminated thread has been joined, or a detached thread has terminated.

The thread ID returned by **pthread_self**() is not the same thing as the kernel thread ID returned by a call to **gettid**(2).

SEE ALSO

pthread_create(3), **pthread_equal**(3), **pthreads**(7)

NAME

sigaction – POSIX signal handling functions.

SYNOPSIS

#include <signal.h>

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

RETURN VALUES

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

ERRORS

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

SEE ALSO

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

NAME

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)

NAME

strcat, strchr, strcmp, strcpy, strdup, strlen, strncat, strncmp, strncpy, strrchr, strstr, strtok − string operations

SYNOPSIS

#include <string.h>

char *strcat(char *dest, const char *src);
Append the string *src* to the string *dest*, returning a pointer *dest*.

char *strchr(const char *s, int c);
Return a pointer to the first occurrence of the character *c* in the string *s*.

int strcmp(const char *s1, const char *s2);
Compare the strings *s1* with *s2*. It returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*.

char *strcpy(char *dest, const char *src);
Copy the string *src* to *dest*, returning a pointer to the start of *dest*.

char *strdup(const char *s);
Return a duplicate of the string *s* in memory allocated using **malloc**(3).

size_t strlen(const char *s);
Return the length of the string *s*.

char *strncat(char *dest, const char *src, size_t n);
Append at most *n* characters from the string *src* to the string *dest*, returning a pointer to *dest*.

int strncmp(const char *s1, const char *s2, size_t n);
Compare at most *n* bytes of the strings *s1* and *s2*. It returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*.

char *strncpy(char *dest, const char *src, size_t n);
Copy at most *n* bytes from string *src* to *dest*, returning a pointer to the start of *dest*.

char *strrchr(const char *s, int c);
Return a pointer to the last occurrence of the character *c* in the string *s*.

char *strstr(const char *haystack, const char *needle);
Find the first occurrence of the substring *needle* in the string *haystack*, returning a pointer to the found substring.

char *strtok(char *s, const char *delim);
Extract tokens from the string *s* that are delimited by one of the bytes in *delim*.

DESCRIPTION

The string functions perform operations on null-terminated strings.