NAME

stat, fstat, lstat, fstatat − get file status

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

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

int stat(const char *restrict pathname,
         struct stat *restrict statbuf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char *restrict pathname,
         struct stat *restrict statbuf);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <sys/stat.h>

int fstatat(int dirfd, const char *restrict pathname,
         struct stat *restrict statbuf, int flags);
```

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

lstat():

```
/* Since glibc 2.20 */ _DEFAULT_SOURCE
   || _XOPEN_SOURCE >= 500
   || /* Since glibc 2.10: */ _POSIX_C_SOURCE >= 200112L
   || /* glibc 2.19 and earlier */ _BSD_SOURCE
```

fstatat():

```
Since glibc 2.10:
   _POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
   _ATFILE_SOURCE
```

DESCRIPTION

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

stat() and **fstatat**() retrieve information about the file pointed to by *pathname*; the differences for **fstatat**() are described below.

lstat() is identical to **stat**(), except that if *pathname* is a symbolic link, then it returns information about the link itself, not the file that the link refers to.

fstat() is identical to **stat**(), except that the file about which information is to be retrieved is specified by the file descriptor *fd*.

The stat structure

All of these system calls return a *stat* structure (see **stat**(3type)).

Note: for performance and simplicity reasons, different fields in the *stat* structure may contain state information from different moments during the execution of the system call. For example, if *st_mode* or *st_uid* is changed by another process by calling **chmod**(2) or **chown**(2), **stat**() might return the old *st_mode* together with the new *st_uid*, or the old *st_uid* together with the new *st_mode*.

fstatat()

The **fstatat**() system call is a more general interface for accessing file information which can still provide exactly the behavior of each of **stat**(), **lstat**(), and **fstat**().

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by **stat**() and **lstat**() for a relative pathname).

If *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relative to the current working directory of the calling process (like **stat**() and **lstat**()).

If *pathname* is absolute, then *dirfd* is ignored.

flags can either be 0, or include one or more of the following flags ORed:

AT_EMPTY_PATH (since Linux 2.6.39)
If *pathname* is an empty string, operate on the file referred to by *dirfd* (which may have been obtained using the **open**(2) **O_PATH** flag). In this case, *dirfd* can refer to any type of file, not just a directory, and the behavior of **fstatat**() is similar to that of **fstat**(). If*dirfd* is **A T_FDCWD**, the call operates on the current working directory. This flag is Linux-specific; define **_GNU_SOURCE** to obtain its definition.

AT_NO_AUTOMOUNT (since Linux 2.6.38)
Don't automount the terminal ("basename") component of *pathname*. Since Linux 3.1 this flag is ignored. Since Linux 4.11 this flag is implied.

AT_SYMLINK_NOFOLLOW
If *pathname* is a symbolic link, do not dereference it: instead return information about the link itself, like **lstat**(). (By default, **fstatat**() dereferences symbolic links, like **stat**().)

See **openat**(2) for an explanation of the need for **fstatat**().

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set to indicate the error.

ERRORS

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

EBADF
fd is not a valid open file descriptor.

EBADF
(**fstatat**()) *pathname* is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EFAULT
Bad address.

EINVAL
(**fstatat**()) Invalid flag specified in *flags*.

ELOOP
Too many symbolic links encountered while traversing the path.

ENAMETOOLONG
pathname is too long.

ENOENT
A component of *pathname* does not exist or is a dangling symbolic link.

ENOENT
pathname is an empty string and **AT_EMPTY_PATH** was not specified in *flags*.

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

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

ENOTDIR
(**fstatat**()) *pathname* is relative and *dirfd* is a file descriptor referring to a file other than a directory.

EOVERFLOW
pathname or *fd* refers to a file whose size, inode number, or number of blocks cannot be represented in, respectively, the types *off_t*, *ino_t*, or *blkcnt_t*. This error can occur when, for example, an application compiled on a 32-bit platform without *−D_FILE_OFFSET_BITS=64* calls **stat**() on a file whose size exceeds *(1<<31)−1* bytes.

VERSIONS
fstatat() was added in Linux 2.6.16; library support was added in glibc 2.4.

STANDARDS
stat(), **fstat**(), **lstat**(): SVr4, 4.3BSD, POSIX.1-2001, POSIX.1.2008.

fstatat(): POSIX.1-2008.

According to POSIX.1-2001, **lstat**() on a symbolic link need return valid information only in the *st_size* field and the file type of the *st_mode* field of the *stat* structure. POSIX.1-2008 tightens the specification, requiring **lstat**() to return valid information in all fields except the mode bits in *st_mode*.

Use of the *st_blocks* and *st_blksize* fields may be less portable. (They were introduced in BSD. The interpretation differs between systems, and possibly on a single system when NFS mounts are involved.)

NOTES
C library/kernel differences
Over time, increases in the size of the *stat* structure have led to three successive versions of **stat**(): *sys_stat*() (slot *__NR_oldstat*), *sys_newstat*() (slot *__NR_stat*), and *sys_stat64()* (slot *__NR_stat64*) on 32-bit platforms such as i386. The first two versions were already present in Linux 1.0 (albeit with different names); the last was added in Linux 2.4. Similar remarks apply for **fstat**() and **lstat**().

The kernel-internal versions of the *stat* structure dealt with by the different versions are, respectively:

__old_kernel_stat
The original structure, with rather narrow fields, and no padding.

stat Larger *st_ino* field and padding added to various parts of the structure to allow for future expansion.

stat64 Even larger *st_ino* field, larger *st_uid* and *st_gid* fields to accommodate the Linux-2.4 expansion of UIDs and GIDs to 32 bits, and various other enlarged fields and further padding in the structure. (Various padding bytes were eventually consumed in Linux 2.6, with the advent of 32-bit device IDs and nanosecond components for the timestamp fields.)

The glibc **stat**() wrapper function hides these details from applications, invoking the most recent version of the system call provided by the kernel, and repacking the returned information if required for old binaries.

On modern 64-bit systems, life is simpler: there is a single **stat**() system call and the kernel deals with a *stat* structure that contains fields of a sufficient size.

The underlying system call employed by the glibc **fstatat**() wrapper function is actually called **fstatat64**() or, on some architectures, **newfstatat**().

EXAMPLES
The following program calls **lstat**() and displays selected fields in the returned *stat* structure.

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/sysmacros.h>
#include <time.h>

int
main(int argc, char *argv[])
{
```

```
        struct stat sb;

        if (argc != 2) {
            fprintf(stderr, "Usage: %s <pathname>\n", argv[0]);
            exit(EXIT_FAILURE);
        }

        if (lstat(argv[1], &sb) == -1) {
            perror("lstat");
            exit(EXIT_FAILURE);
        }

        printf("ID of containing device:  [%x,%x]\n",
               major(sb.st_dev),
               minor(sb.st_dev));

        printf("File type:                ");

        switch (sb.st_mode & S_IFMT) {
        case S_IFBLK:  printf("block device\n");            break;
        case S_IFCHR:  printf("character device\n");        break;
        case S_IFDIR:  printf("directory\n");               break;
        case S_IFIFO:  printf("FIFO/pipe\n");               break;
        case S_IFLNK:  printf("symlink\n");                 break;
        case S_IFREG:  printf("regular file\n");            break;
        case S_IFSOCK: printf("socket\n");                  break;
        default:       printf("unknown?\n");                break;
        }

        printf("I-node number:            %ju\n", (uintmax_t) sb.st_ino);

        printf("Mode:                     %jo (octal)\n",
               (uintmax_t) sb.st_mode);

        printf("Link count:               %ju\n", (uintmax_t) sb.st_nlink);
        printf("Ownership:                UID=%ju   GID=%ju\n",
               (uintmax_t) sb.st_uid, (uintmax_t) sb.st_gid);

        printf("Preferred I/O block size: %jd bytes\n",
               (intmax_t) sb.st_blksize);
        printf("File size:                %jd bytes\n",
               (intmax_t) sb.st_size);
        printf("Blocks allocated:         %jd\n",
               (intmax_t) sb.st_blocks);

        printf("Last status change:       %s", ctime(&sb.st_ctime));
        printf("Last file access:         %s", ctime(&sb.st_atime));
        printf("Last file modification:   %s", ctime(&sb.st_mtime));

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

ls(1), **stat**(1), **access**(2), **chmod**(2), **chown**(2), **readlink**(2), **statx**(2), **utime**(2), **stat**(3type), **capabilities**(7), **inode**(7), **symlink**(7)