NAME

stat, fstat, lstat, fstatat – get file status

SYNOPSIS

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

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

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

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

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

lstat():

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

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 it 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, 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;        /* File type and mode */
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;     /* Block size for filesystem I/O */
blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

/* Since Linux 2.6, the kernel supports nanosecond
precision for the following timestamp fields.
```

```
    For the details before Linux 2.6, see NOTES. */

    struct timespec st_atim;  /* Time of last access */
    struct timespec st_mtim;  /* Time of last modification */
    struct timespec st_ctim;  /* Time of last status change */

    #define st_atime st_atim.tv_sec      /* Backward compatibility */
    #define st_mtime st_mtim.tv_sec
    #define st_ctime st_ctim.tv_sec
    };
```

Note: the order of fields in the *stat* structure varies somewhat across architectures. In addition, the definition above does not show the padding bytes that may be present between some fields on various architectures. Consult the glibc and kernel source code if you need to know the details.

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*.

The fields in the *stat* structure are as follows:

st_dev This field describes the device on which this file resides. (The major(3) and minor(3) macros may be useful to decompose the device ID in this field.)

st_ino This field contains the file's inode number.

st_mode
This field contains the file type and mode. See inode(7) for further information.

st_nlink
This field contains the number of hard links to the file.

st_uid This field contains the user ID of the owner of the file.

st_gid This field contains the ID of the group owner of the file.

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

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

st_blksize
This field gives the "preferred" block size for efficient filesystem I/O.

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

st_atime
This is the file's last access timestamp.

st_mtime
This is the file's last modification timestamp.

st_ctime
This is the file's last status change timestamp.

For further information on the above fields, see inode(7).

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 **AT_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* if it is a directory that is an automount point. This allows the caller to gather attributes of an automount point (rather than the location it would mount). Since Linux 4.14, also don't instantiate a nonexistent name in an on-demand directory such as used for automounter indirect maps. This flag can be used in tools that scan directories to prevent mass-automounting of a directory of automount points. The **AT_NO_AUTOMOUNT** flag has no effect if the mount point has already been mounted over. This flag is Linux-specific; define **_GNU_SOURCE** to obtain its definition. Both **stat**() and **lstat**() act as though **AT_NO_AUTOMOUNT** was set.

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 appropriately.

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.

EFAULT
: Bad address.

ELOOP
: Too many symbolic links encountered while traversing the path.

ENAMETOOLONG
: *pathname* is too long.

ENOENT
: A component of *pathname* does not exist, or *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.

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.

The following additional errors can occur for **fstatat**():

EBADF
dirfd is not a valid file descriptor.

EINVAL
Invalid flag specified in *flags*.

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

VERSIONS

fstatat() was added to Linux in kernel 2.6.16; library support was added to glibc in version 2.4.

CONFORMING TO

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

Timestamp fields

Older kernels and older standards did not support nanosecond timestamp fields. Instead, there were three timestamp fields—*st_atime*, *st_mtime*, and *st_ctime*—typed as *time_t* that recorded timestamps with one-second precision.

Since kernel 2.5.48, the *stat* structure supports nanosecond resolution for the three file timestamp fields. The nanosecond components of each timestamp are available via names of the form *st_atim.tv_nsec*, if suitable feature test macros are defined. Nanosecond timestamps were standardized in POSIX.1-2008, and, starting with version 2.12, glibc exposes the nanosecond component names if **_POSIX_C_SOURCE** is defined with the value 200809L or greater, or **_XOPEN_SOURCE** is defined with the value 700 or greater. Up to and including glibc 2.19, the definitions of the nanoseconds components are also defined if **_BSD_SOURCE** or **_SVID_SOURCE** is defined. If none of the aforementioned macros are defined, then the nanosecond values are exposed with names of the form *st_atimensec*.

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**().

EXAMPLE

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

```
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/sysmacros.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:  [%lx,%lx]\n",
          (long) major(sb.st_dev), (long) 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:            %ld\n", (long) sb.st_ino);

printf("Mode:                     %lo (octal)\n",
(unsigned long) sb.st_mode);

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

printf("Preferred I/O block size: %ld bytes\n",
(long) sb.st_blksize);
printf("File size:                %lld bytes\n",
(long long) sb.st_size);
printf("Blocks allocated:         %lld\n",
```

```
        (long long) 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), utime(2), capabilities(7), inode(7), symlink(7)

COLOPHON

This page is part of release 4.16 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/.