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

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

int fstatat(int dirfd, const char *pathname, struct stat *buf,
 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 || _XOPEN_SOURCE && _XOPEN_SOURCE_EXTENDED
|| /* Since glibc 2.10: */ _POSIX_C_SOURCE >= 200112L

fstatat():
Since glibc 2.10:
_XOPEN_SOURCE >= 700 || _POSIX_C_SOURCE >= 200809L
Before glibc 2.10:
_ATFILE_SOURCE

DESCRIPTION

These functions return information about a file, in the buffer pointed to by *stat*. 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*.

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 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 the glibc and kernel source code if you need to know the details.

The *st_dev* 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.)

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 symbolic link is the length of the pathname it contains, without a terminating 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 filesystem I/O. (Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.)

Not all of the Linux filesystems implement all of the time fields. Some filesystem types allow mounting in such a way that file and/or directory accesses do not cause an update of the *st_atime* field. (See*no atime*, *nodiratime*, and *relatime* in mount(8), and related information in mount(2).) In addition, *st_atime* is not updated if a file is opened with the **O_NOATIME**; see open(2).

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* c hanged 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 mask values are defined for the file type component of the *st_mode* field:

S_IFMT	0170000	bit mask for the file type bit fields
S_IFSOCK	0140000	socket
S_IFLNK	0120000	symbolic link
S_IFREG	0100000	regular file
S_IFBLK	0060000	block device
S_IFDIR	0040000	directory
S_IFCHR	0020000	character device
S_IFIFO	0010000	FIFO

Thus, to test for a regular file (for example), one could write:

```
stat(pathname, &sb);
if ((sb.st_mode & S_IFMT) == S_IFREG) {
```

```
/* Handle regular file */
}
```

Because tests of the above form are common, additional macros are defined by POSIX to allow the test of the file type in *st_mode* to be written more concisely:

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.)

The preceding code snippet could thus be rewritten as:

```
stat(pathname, &sb);
if (S_ISREG(sb.st_mode)) {
/* Handle regular file */
}
```

The definitions of most of the above file type test macros are provided if any of the following feature test macros is defined: **_BSD_SOURCE** (in glibc 2.19 and earlier), **_SVID_SOURCE** (in glibc 2.19 and earlier), or **_DEFAULT_SOURCE** (in glibc 2.20 and later). In addition, definitions of all of the above macros except **S_IFSOCK** and **S_ISSOCK**() are provided if **_XOPEN_SOURCE** is defined. The definition of **S_IFSOCK** can also be exposed by defining **_XOPEN_SOURCE** with a value of 500 or greater.

The definition of **S_ISSOCK**() is exposed if any of the following feature test macros is defined: **_BSD_SOURCE** (in glibc 2.19 and earlier), **_DEFAULT_SOURCE** (in glibc 2.20 and later), **_XOPEN_SOURCE** with a value of 500 or greater, or **_POSIX_C_SOURCE** with a value of 200112L or greater.

The following mask values are defined for the file permissions component of the *st_mode* field:

S_ISUID	0004000	set-user-ID bit
S_ISGID	0002000	set-group-ID bit (see below)
S_ISVTX	0001000	sticky bit (see below)
S_IRWXU	00700	mask for file owner permissions
S_IRUSR	00400	owner has read permission
S_IWUSR	00200	owner has write permission
S_IXUSR	00100	owner has execute permission
S_IRWXG	00070	mask for group permissions
S_IRGRP	00040	group has read permission
S_IWGRP	00020	group has write permission
S_IXGRP	00010	group has execute permission
S_IRWXO	00007	mask for permissions for others (not in group)
S_IROTH	00004	others have read permission
S_IWOTH	00002	others have write permission
S_IXOTH	00001	others have execute permission

The set-group-ID bit (**S_ISGID**) has several special uses. For a directory, it indicates that BSD semantics is to be used for that directory: files created there inherit their group ID from the

directory, not from the effective group ID of the creating process, and directories created there will also get the **S_ISGID** bit set. For a file that does not have the group execution bit (**S_IXGRP**) set, the set-group-ID bit indicates mandatory file/record locking.

The sticky bit (**S_ISVTX**) on a directory means that a file in that directory can be renamed or deleted only by the owner of the file, by the owner of the directory, and by a privileged process.

fstatat()

The **fstatat**() system call operates in exactly the same way as **stat**(), except for the differences described here.

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**() 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**()).

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). If *dirfd* is **AT_FDCWD**, the call operates on the current working directory. In this case,*dirfd* can refer to an y type of file, not just a 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). 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.

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 sym bolic 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 bad.

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.

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 component 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 permission 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.) If you need to obtain the definition of the *blkcnt_t* or *blksize_t* types from *<sys/stat.h>*, then define **_XOPEN_SOURCE** with the value 500 or greater (before including *any* header files).

POSIX.1-1990 did not describe the **S_IFMT**, **S_IFSOCK**, **S_IFLNK**, **S_IFREG**, **S_IFBLK**, **S_IFDIR**, **S_IFCHR**, **S_IFIFO**, **S_ISVTX** constants, but instead demanded the use of the macros **S_ISDIR**(), and so on. The **S_IF*** constants are present in POSIX.1-2001 and later.

The **S_ISLNK**() and **S_ISSOCK**() macros are not in POSIX.1-1996, but both are present in POSIX.1-2001; the former is from SVID 4, the latter from SUSv2.

UNIX V7 (and later systems) had **S_IREAD**, **S_IWRITE**, **S_IEXEC**, where POSIX prescribes the synonyms **S_IRUSR**, **S_IWUSR**, **S_IXUSR**.

Other systems

Values that have been (or are) in use on various systems:

hex	name	ls	octal	description
f000	S_IFMT		170000	mask for file type

0000			000000	SCO out-of-service inode; BSD unknown type; SVID-v2 and XPG2 have both 0 and 0100000 for ordinary file
1000	S_IFIFO	p|	010000	FIFO (named pipe)
2000	S_IFCHR	c	020000	character special (V7)
3000	S_IFMPC		030000	multiplexed character special (V7)
4000	S_IFDIR	d/	040000	directory (V7)
5000	S_IFNAM		050000	XENIX named special file with two subtypes, distinguished by *st_rdev* values 1, 2
0001	S_INSEM	s	000001	XENIX semaphore subtype of IFNAM
0002	S_INSHD	m	000002	XENIX shared data subtype of IFNAM
6000	S_IFBLK	b	060000	block special (V7)
7000	S_IFMPB		070000	multiplexed block special (V7)
8000	S_IFREG	-	100000	regular (V7)
9000	S_IFCMP		110000	VxFS compressed
9000	S_IFNWK	n	110000	network special (HP-UX)
a000	S_IFLNK	l@	120000	symbolic link (BSD)
b000	S_IFSHAD		130000	Solaris shadow inode for ACL (not seen by user space)
c000	S_IFSOCK	s=	140000	socket (BSD; also S_IFSOC on VxFS)
d000	S_IFDOOR	D>	150000	Solaris door
e000	S_IFWHT	w%	160000	BSD whiteout (not used for inode)
0200	S_ISVTX		001000	sticky bit: save swapped text even after use (V7) reserved (SVID-v2) On nondirectories: don't cache this file (SunOS) On directories: restricted deletion flag (SVID-v4.2)
0400	S_ISGID		002000	set-group-ID on execution (V7) for directories: use BSD semantics for propagation of GID
0400	S_ENFMT		002000	System V file locking enforcement (shared with S_ISGID)
0800	S_ISUID		004000	set-user-ID on execution (V7)
0800	S_CDF		004000	directory is a context dependent file (HP-UX)

A sticky command appeared in Version 32V AT&T UNIX.

NOTES

On Linux, **lstat**() will generally not trigger automounter action, whereas **stat**() will (but see fstatat(2)).

For most files under the */proc* directory, **stat**() does not return the file size in the *st_size* field; instead the field is returned with the value 0.

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 the **_BSD_SOURCE** or **_SVID_SOURCE** feature test macro is defined.

Nanosecond timestamps are nowadays standardized, starting with POSIX.1-2008, and, starting with version 2.12, glibc also 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. If none of the aforementioned macros are defined, then the nanosecond values are exposed with names of the form *st_atimensec*.

Nanosecond timestamps are supported on XFS, JFS, Btrfs, and ext4 (since Linux 2.6.23). Nanosecond timestamps are not supported in ext2, ext3, and Reiserfs. On filesystems that do not support subsecond timestamps, the nanosecond fields are returned with the value 0.

Underlying kernel interface

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()* (new in kernel 2.4; slot *__NR_stat64*). 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. Similar remarks apply for **fstat**() and **lstat**().

The underlying system call employed by the glibc **fstatat**() wrapper function is actually called **fstatat64**().

EXAMPLE

The following program calls **stat**() 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>

int
main(int argc, char *argv[])
{
struct stat sb;

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

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

printf(File type: );

switch (sb.st_mode & S_IFMT) {
case S_IFBLK: printf(block devicen); break;
case S_IFCHR: printf(character devicen); break;
case S_IFDIR: printf(directoryn); break;
case S_IFIFO: printf(FIFO/pipen); break;
case S_IFLNK: printf(symlinkn); break;
case S_IFREG: printf(regular filen); break;
case S_IFSOCK: printf(socketn); break;
default: printf(unknown?n); break;
}

printf(I-node number: %ldn, (long) sb.st_ino);

printf(Mode: %lo (octal)n,
```

```
(unsigned long) sb.st_mode);

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

printf(Preferred I/O block size: %ld bytesn,
(long) sb.st_blksize);
printf(File size: %lld bytesn,
(long long) sb.st_size);
printf(Blocks allocated: %lldn,
(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), symlink(7)

COLOPHON

This page is part of release 3.74 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 http://www.kernel.org/doc/man-pages/.