NAME

inode − file inode information

DESCRIPTION

Each file has an inode containing metadata about the file. An application can retrieve this metadata using **stat**(2) (or related calls), which returns a *stat* structure, or **statx**(2), which returns a *statx* structure.

The following is a list of the information typically found in, or associated with, the file inode, with the names of the corresponding structure fields returned by **stat**(2) and **statx**(2):

Device where inode resides

stat.st_dev; *statx.stx_dev_minor* and *statx.stx_dev_major*

Each inode (as well as the associated file) resides in a filesystem that is hosted on a device. That device is identified by the combination of its major ID (which identifies the general class of device) and minor ID (which identifies a specific instance in the general class).

Inode number

stat.st_ino; *statx.stx_ino*

Each file in a filesystem has a unique inode number. Inode numbers are guaranteed to be unique only within a filesystem (i.e., the same inode numbers may be used by different filesystems, which is the reason that hard links may not cross filesystem boundaries). This field contains the file's inode number.

File type and mode

stat.st_mode; *statx.stx_mode*

See the discussion of file type and mode, below.

Link count

stat.st_nlink; *statx.stx_nlink*

This field contains the number of hard links to the file. Additional links to an existing file are created using **link**(2).

User ID

st_uid stat.st_uid; *statx.stx_uid*

This field records the user ID of the owner of the file. For newly created files, the file user ID is the effective user ID of the creating process. The user ID of a file can be changed using **chown**(2).

Group ID

stat.st_gid; *statx.stx_gid*

The inode records the ID of the group owner of the file. For newly created files, the file group ID is either the group ID of the parent directory or the effective group ID of the creating process, depending on whether or not the set-group-ID bit is set on the parent directory (see below). The group ID of a file can be changed using **chown**(2).

Device represented by this inode

stat.st_rdev; *statx.stx_rdev_minor* and *statx.stx_rdev_major*

If this file (inode) represents a device, then the inode records the major and minor ID of that device.

File size

stat.st_size; *statx.stx_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.

Preferred block size for I/O

stat.st_blksize; *statx.stx_blksize*

This field gives the "preferred" blocksize for efficient filesystem I/O. (Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.)

Number of blocks allocated to the file
stat.st_blocks; *statx.stx_size*

This 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 POSIX.1 standard notes that the unit for the *st_blocks* member of the *stat* structure is not defined by the standard. On many implementations it is 512 bytes; on a few systems, a different unit is used, such as 1024. Furthermore, the unit may differ on a per-filesystem basis.

Last access timestamp (atime)
stat.st_atime; *statx.stx_atime*

This is the file's last access timestamp. It 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 interfaces, such as **mmap**(2), may or may not update the atime timestamp

Some filesystem types allow mounting in such a way that file and/or directory accesses do not cause an update of the atime timestamp. (See *noatime*, *nodiratime*, and *relatime* in **mount**(8), and related information in **mount**(2).) In addition, the atime timestamp is not updated if a file is opened with the **O_NOATIME** flag; see **open**(2).

File creation (birth) timestamp (btime)
(not returned in the *stat* structure); *statx.stx_btime*

The file's creation timestamp. This is set on file creation and not changed subsequently.

The btime timestamp was not historically present on UNIX systems and is not currently supported by most Linux filesystems.

Last modification timestamp (mtime)
stat.st_mtime; *statx.stx_mtime*

This is the file's last modification timestamp. It is changed by file modifications, for example, by **mknod**(2), **truncate**(2), **utime**(2), and **write**(2) (of more than zero bytes). Moreover, the mtime timestamp of a directory is changed by the creation or deletion of files in that directory. The mtime timestamp is *not* changed for changes in owner, group, hard link count, or mode.

Last status change timestamp (ctime)
stat.st_ctime; *statx.stx_ctime*

This is the file's last status change timestamp. It is changed by writing or by setting inode information (i.e., owner, group, link count, mode, etc.).

The timestamp fields report time measured with a zero point at the *Epoch*, 1970-01-01 00:00:00 +0000, UTC (see **time**(7)).

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. In order to return timestamps with nanosecond precision, the timestamp fields in the *stat* and *statx* structures are defined as structures that include a nanosecond component. See **stat**(2) and **statx**(2) for details. On filesystems that do not support subsecond timestamps, the nanosecond fields in the *stat* and *statx* structures are returned with the value 0.

The file type and mode

The *stat.st_mode* field (for **statx**(2), the *statx.stx_mode* field) contains the file type and mode.

POSIX refers to the *stat.st_mode* bits corresponding to the mask **S_IFMT** (see below) as the *file type*, the 12 bits corresponding to the mask 07777 as the *file mode bits* and the least significant 9 bits (0777) as the *file permission bits*.

The following mask values are defined for the file type:

S_IFMT	0170000	bit mask for the file type bit field
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 either by defining **_XOPEN_SOURCE** with a value of 500 or greater or (since glibc 2.24) by defining both **_XOPEN_SOURCE** and **_XOPEN_SOURCE_EXTENDED**.

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, **_POSIX_C_SOURCE** with a value of 200112L or greater, or (since glibc 2.24) by defining both **_XOPEN_SOURCE** and **_XOPEN_SOURCE_EXTENDED**.

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

S_ISUID	04000	set-user-ID bit (see **execve**(2))
S_ISGID	02000	set-group-ID bit (see below)
S_ISVTX	01000	sticky bit (see below)
S_IRWXU	00700	owner has read, write, and execute permission
S_IRUSR	00400	owner has read permission

S_IWUSR	00200	owner has write permission
S_IXUSR	00100	owner has execute permission
S_IRWXG	00070	group has read, write, and execute permission
S_IRGRP	00040	group has read permission
S_IWGRP	00020	group has write permission
S_IXGRP	00010	group has execute permission
S_IRWXO	00007	others (not in group) have read, write, and execute permission
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 are 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 an executable file, the set-group-ID bit causes the effective group ID of a process that executes the file to change as described in **execve**(2). 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.

STANDARDS

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**, and **S_ISVTX** constants, but instead specified 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 were 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**, and where POSIX prescribes the synonyms **S_IRUSR**, **S_IWUSR**, and **S_IXUSR**.

NOTES

For pseudofiles that are autogenerated by the kernel, the file size (*stat.st_size*; *statx.stx_size*) reported by the kernel is not accurate. For example, the value 0 is returned for many files under the */proc* directory, while various files under */sys* report a size of 4096 bytes, even though the file content is smaller. For such files, one should simply try to read as many bytes as possible (and append '\0' to the returned buffer if it is to be interpreted as a string).

SEE ALSO

stat(1), **stat**(2), **statx**(2), **symlink**(7)