Workshop D

Operating Systems Programming – 300698

1 Introduction

In this workshop you will investigate file I/O and file copy operations.

2 Specification

Following on from the demonstration program `cp1.c` presented in the lecture, we will make a series of modifications to the program.

Firstly, what happens when the `cp1.c` program is asked to copy a file onto itself, i.e. `cp1 input input`? Is this what you expect? Modify the program to do something more sensible! As a hint, two files are the same if they are on the same device and have the same i-node number (which `stat()` can give you), simply comparing the names is not enough.

Secondly, a real copy program will assign the same file permissions to the destination as were on the source, modify your answer to the last part to do this.

Thirdly, real copy programs allow the second argument to be a directory, modify the answer to the last part to include this functionality. You should allocate the space for the new name dynamically.

Fourthly, your program should abort the copy if the destination file exists, or if the destination is a directory that a file with the same name exists in the directory.

3 Marking Scheme

The following functionality items will be considered when evaluating how much of the specification is implemented:

- correct collection of source file information
- correct collection of destination file information
- correct test of files being the same
- something sensible done when files are the same
- correct permissions assigned to destination
- correct detection of destination being a directory
- correct allocation of memory
- correct construction of destination file name
- correct freeing of allocated memory
- correctly aborting when destination exists

The following rubric, taken from the learning guide (with zero weighted criteria removed), will be used to evaluate submissions.

CRITERIA (Weighting)	Unsatisfactory (0%)	Poor (25%)	Good (50%)	Very good (75%)	Excellent (100%)
Readability (10%)	Code is unreadable.	The code is poorly organized and very difficult to read.	The code is readable only by someone who knows what it is supposed to be doing.	The code is fairly easy to read.	The code is exceptionally well organized and very easy to follow.
Documentation (10%)	No documentation provided.	The documentation is simply comments embedded in the code and does not help the reader understand the code.	The documentation is simply comments embedded in the code with some simple header comments separating routines.	The documentation consists of embedded comment and some simple header documentation that is somewhat useful in understanding the code.	The documentation is well written and clearly explains what the code is accomplishing and how.
Error/Exception Handling (20%)	Completed functional requirements handle no error/exception conditions.	Completed functional requirements handle obvious error/ exception conditions, or makes no distinction between benign and fatal errors/exceptions.	Completed functional requirements handle some error/exception conditions, most errors/exceptions correctly categorised as benign or fatal.	Completed functional requirements handle most error/exception conditions, most errors/exceptions correctly categorised as benign or fatal.	Completed functional requirements handle all error/exception conditions, with all errors/exceptions correctly categorised as benign or fatal.
Specifications (60%)	Program produces no results.	The program is producing incorrect results.	The program produces correct results but does not display them correctly.	The program works and produces the correct results and displays them correctly. It also meets most of the other specifications.	The program works and meets all of the specifications.

4 Sample Code

4.1 cp1.c

```
#include        <stdio.h>
#include        <unistd.h>
#include        <fcntl.h>

#define BUFFERSIZE      4096
#define COPYMODE        0644

void oops(char *, char *);

main(int ac, char *av[])
{
        int     in_fd, out_fd, n_chars;
        char    buf[BUFFERSIZE];

        if ( ac != 3 ){
                fprintf( stderr, "usage: %s source destination\n", *av);
                exit(1);
        }

        if ( (in_fd=open(av[1], O_RDONLY)) == -1 )
                oops("Cannot open ", av[1]);

        if ( (out_fd=creat( av[2], COPYMODE)) == -1 )
                oops( "Cannot creat", av[2]);


        while ( (n_chars = read(in_fd , buf, BUFFERSIZE)) > 0 )
                if ( write( out_fd, buf, n_chars ) != n_chars )
                        oops("Write error to ", av[2]);
        if ( n_chars == -1 )
                oops("Read error from ", av[1]);


        if ( close(in_fd) == -1 || close(out_fd) == -1 )
                oops("Error closing files","");
}

void oops(char *s1, char *s2)
{
        fprintf(stderr,"Error: %s ", s1);
        perror(s2);
        exit(1);
}
```

5 Supplementary Materials

The material on the following pages is an extract of the linux system documentation and may prove useful in implementing this Workshop. These manual pages are taken from the Linux *man-pages* Project available at: `http://www.kernel.org/doc/man-pages/`.

NAME

open, creat − open and possibly create a file or device

SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char * *pathname***, int** *flags***);**
int open(const char * *pathname***, int** *flags***, mode_t** *mode***);**
int creat(const char * *pathname***, mode_t** *mode***);**

DESCRIPTION

Given a *pathname* for a file, **open**() returns a file descriptor, a small, non-negative integer for use in subsequent system calls (**read**(2), **write**(2), **lseek**(2), **fcntl**(2), etc.). The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

The new file descriptor is set to remain open across an **execve**(2) (i.e., the **FD_CLOEXEC** file descriptor flag described in **fcntl**(2) is initially disabled). The file offset is set to the beginning of the file (see **lseek**(2)).

A call to **open**() creates a new *open file description*, an entry in the system-wide table of open files. This entry records the file offset and the file status flags (modifiable via the **fcntl**() **F_SETFL** operation). A file descriptor is a reference to one of these entries; this reference is unaffected if *pathname* is subsequently removed or modified to refer to a different file. The new open file description is initially not shared with any other process, but sharing may arise via **fork**(2).

The parameter *flags* must include one of the following *access modes*: **O_RDONLY**, **O_WRONLY**, or **O_RDWR.** These request opening the file read-only, write-only, or read/write, respectively.

In addition, zero or more file creation flags and file status flags can be bitwise-*or*'d in *flags*. The *file creation flags* are **O_CREAT**, **O_EXCL**, **O_NOCTTY**, and **O_TRUNC**. The *file status flags* are all of the remaining flags listed below. The distinction between these two groups of flags is that the file status flags can be retrieved and (in some cases) modified using **fcntl**(2). The full list of file creation flags and file status flags is as follows:

O_APPEND
The file is opened in append mode. Before each **write**(), the file offset is positioned at the end of the file, as if with **lseek**(). **O_APPEND** may lead to corrupted files on NFS file systems if more than one process appends data to a file at once. This is because NFS does not support appending to a file, so the client kernel has to simulate it, which can't be done without a race condition.

O_ASYNC
Enable signal-driven I/O: generate a signal (SIGIO by default, but this can be changed via **fcntl**(2)) when input or output becomes possible on this file descriptor. This feature is only available for terminals, pseudo-terminals, sockets, and (since Linux 2.6) pipes and FIFOs. See **fcntl**(2) for further details.

O_CREAT
If the file does not exist it will be created. The owner (user ID) of the file is set to the effective user ID of the process. The group ownership (group ID) is set either to the effective group ID of the process or to the group ID of the parent directory (depending on filesystem type and mount options, and the mode of the parent directory, see, e.g., the mount options *bsdgroups* and *sysvgroups* of the ext2 filesystem, as described in **mount**(8)).

O_DIRECT
Try to minimize cache effects of the I/O to and from this file. In general this will degrade performance, but it is useful in special situations, such as when applications do their own caching. File I/O is done directly to/from user space buffers. The I/O is synchronous, i.e., at the completion of a **read**(2) or **write**(2), data is guaranteed to have been transferred. Under Linux 2.4 transfer sizes,

and the alignment of user buffer and file offset must all be multiples of the logical block size of the file system. Under Linux 2.6 alignment must fit the block size of the device.

A semantically similar (but deprecated) interface for block devices is described in **raw**(8).

O_DIRECTORY
If *pathname* is not a directory, cause the open to fail. This flag is Linux-specific, and was added in kernel version 2.1.126, to avoid denial-of-service problems if **opendir**(3) is called on a FIFO or tape device, but should not be used outside of the implementation of **opendir**.

O_EXCL
When used with **O_CREAT**, if the file already exists it is an error and the **open**() will fail. In this context, a symbolic link exists, regardless of where it points to. **O_EXCL** is broken on NFS file systems; programs which rely on it for performing locking tasks will contain a race condition. The solution for performing atomic file locking using a lockfile is to create a unique file on the same file system (e.g., incorporating hostname and pid), use **link**(2) to make a link to the lockfile. If **link**() returns 0, the lock is successful. Otherwise, use **stat**(2) on the unique file to check if its link count has increased to 2, in which case the lock is also successful.

O_LARGEFILE
(LFS) Allow files whose sizes cannot be represented in an *off_t* (but can be represented in an *off64_t*) to be opened.

O_NOATIME
(Since Linux 2.6.8) Do not update the file last access time (st_atime in the inode) when the file is **read**(2). This flag is intended for use by indexing or backup programs, where its use can significantly reduce the amount of disk activity. This flag may not be effective on all filesystems. One example is NFS, where the server maintains the access time.

O_NOCTTY
If *pathname* refers to a terminal device — see **tty**(4) — it will not become the process's controlling terminal even if the process does not have one.

O_NOFOLLOW
If *pathname* is a symbolic link, then the open fails. This is a FreeBSD extension, which was added to Linux in version 2.1.126. Symbolic links in earlier components of the pathname will still be followed.

O_NONBLOCK or **O_NDELAY**
When possible, the file is opened in non-blocking mode. Neither the **open**() nor any subsequent operations on the file descriptor which is returned will cause the calling process to wait. For the handling of FIFOs (named pipes), see also **fifo**(7). For a discussion of the effect of **O_NONBLOCK** in conjunction with mandatory file locks and with file leases, see **fcntl**(2).

O_SYNC
The file is opened for synchronous I/O. Any **write**()s on the resulting file descriptor will block the calling process until the data has been physically written to the underlying hardware. *But see RESTRICTIONS below*.

O_TRUNC
If the file already exists and is a regular file and the open mode allows writing (i.e., is O_RDWR or O_WRONLY) it will be truncated to length 0. If the file is a FIFO or terminal device file, the O_TRUNC flag is ignored. Otherwise the effect of O_TRUNC is unspecified.

Some of these optional flags can be altered using **fcntl**() after the file has been opened.

The argument *mode* specifies the permissions to use in case a new file is created. It is modified by the process's **umask** in the usual way: the permissions of the created file are **(mode & ~umask)**. Note that this mode only applies to future accesses of the newly created file; the **open**() call that creates a read-only file

may well return a read/write file descriptor.

The following symbolic constants are provided for *mode*:

S_IRWXU
00700 user (file owner) has read, write and execute permission

S_IRUSR
00400 user has read permission

S_IWUSR
00200 user has write permission

S_IXUSR
00100 user 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 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

mode must be specified when **O_CREAT** is in the *flags*, and is ignored otherwise.

creat() is equivalent to **open**() with *flags* equal to **O_CREAT|O_WRONLY|O_TRUNC**.

RETURN VALUE

open() and **creat**() return the new file descriptor, or −1 if an error occurred (in which case, *errno* is set appropriately).

NOTES

Note that **open**() can open device special files, but **creat**() cannot create them; use **mknod**(2) instead.

On NFS file systems with UID mapping enabled, **open**() may return a file descriptor but e.g. **read**(2) requests are denied with **EACCES**. This is because the client performs **open**() by checking the permissions, but UID mapping is performed by the server upon read and write requests.

If the file is newly created, its st_atime, st_ctime, st_mtime fields (respectively, time of last access, time of last status change, and time of last modification; see **stat**(2)) are set to the current time, and so are the st_ctime and st_mtime fields of the parent directory. Otherwise, if the file is modified because of the O_TRUNC flag, its st_ctime and st_mtime fields are set to the current time.

ERRORS

EACCES
The requested access to the file is not allowed, or search permission is denied for one of the directories in the path prefix of *pathname*, or the file did not exist yet and write access to the parent directory is not allowed. (See also **path_resolution**(2).)

EEXIST
pathname already exists and **O_CREAT** and **O_EXCL** were used.

EFAULT
pathname points outside your accessible address space.

EISDIR
pathname refers to a directory and the access requested involved writing (that is, **O_WRONLY** or **O_RDWR** is set).

ELOOP
Too many symbolic links were encountered in resolving *pathname*, or **O_NOFOLLOW** was specified but *pathname* was a symbolic link.

EMFILE
The process already has the maximum number of files open.

ENAMETOOLONG
pathname was too long.

ENFILE
The system limit on the total number of open files has been reached.

ENODEV
pathname refers to a device special file and no corresponding device exists. (This is a Linux kernel bug; in this situation ENXIO must be returned.)

ENOENT
O_CREAT is not set and the named file does not exist. Or, a directory component in *pathname* does not exist or is a dangling symbolic link.

ENOMEM
Insufficient kernel memory was available.

ENOSPC
pathname was to be created but the device containing *pathname* has no room for the new file.

ENOTDIR
A component used as a directory in *pathname* is not, in fact, a directory, or **O_DIRECTORY** was specified and *pathname* was not a directory.

ENXIO
O_NONBLOCK | O_WRONLY is set, the named file is a FIFO and no process has the file open for reading. Or, the file is a device special file and no corresponding device exists.

EOVERFLOW
pathname refers to a regular file, too large to be opened; see O_LARGEFILE above.

EPERM
The **O_NOATIME** flag was specified, but the effective user ID of the caller did not match the owner of the file and the caller was not privileged (**CAP_FOWNER**).

EROFS
pathname refers to a file on a read-only filesystem and write access was requested.

ETXTBSY
pathname refers to an executable image which is currently being executed and write access was requested.

EWOULDBLOCK
The **O_NONBLOCK** flag was specified, and an incompatible lease was held on the file (see **fcntl**(2)).

NOTE

Under Linux, the O_NONBLOCK flag indicates that one wants to open but does not necessarily have the intention to read or write. This is typically used to open devices in order to get a file descriptor for use with **ioctl**(2).

CONFORMING TO

SVr4, 4.3BSD, POSIX.1-2001. The **O_NOATIME**, **O_NOFOLLOW**, and **O_DIRECTORY** flags are Linux-specific. One may have to define the **_GNU_SOURCE** macro to get their definitions.

The (undefined) effect of **O_RDONLY | O_TRUNC** varies among implementations. On many systems the file is actually truncated.

The **O_DIRECT** flag was introduced in SGI IRIX, where it has alignment restrictions similar to those of Linux 2.4. IRIX has also a fcntl(2) call to query appropriate alignments, and sizes. FreeBSD 4.x introduced a flag of same name, but without alignment restrictions. Support was added under Linux in kernel version 2.4.10. Older Linux kernels simply ignore this flag. One may have to define the **_GNU_SOURCE** macro to get its definition.

BUGS

"The thing that has always disturbed me about O_DIRECT is that the whole interface is just stupid, and was probably designed by a deranged monkey on some serious mind-controlling substances." — Linus

Currently, it is not possible to enable signal-driven I/O by specifying **O_ASYNC** when calling **open**(); use **fcntl**(2) to enable this flag.

RESTRICTIONS

There are many infelicities in the protocol underlying NFS, affecting amongst others **O_SYNC** and **O_NDELAY**.

POSIX provides for three different variants of synchronised I/O, corresponding to the flags **O_SYNC**, **O_DSYNC** and **O_RSYNC**. Currently (2.1.130) these are all synonymous under Linux.

SEE ALSO

close(2), **dup**(2), **fcntl**(2), **link**(2), **lseek**(2), **mknod**(2), **mount**(2), **mmap**(2), **openat**(2), **path_resolution**(2), **read**(2), **socket**(2), **stat**(2), **umask**(2), **unlink**(2), **write**(2), **fopen**(3), **fifo**(7), **feature_test_macros**(7)

NAME

close − close a file descriptor

SYNOPSIS

#include <unistd.h>

int close(int *fd***);**

DESCRIPTION

close() closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks (see **fcntl**(2)) held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).

If *fd* is the last copy of a particular file descriptor the resources associated with it are freed; if the descriptor was the last reference to a file which has been removed using **unlink**(2) the file is deleted.

RETURN VALUE

close() returns zero on success. On error, −1 is returned, and *errno* is set appropriately.

ERRORS

EBADF
fd isn't a valid open file descriptor.

EINTR
The **close**() call was interrupted by a signal.

EIO An I/O error occurred.

CONFORMING TO

SVr4, 4.3BSD, POSIX.1-2001.

NOTES

Not checking the return value of **close**() is a common but nevertheless serious programming error. It is quite possible that errors on a previous **write**(2) operation are first reported at the final **close**(). Not checking the return value when closing the file may lead to silent loss of data. This can especially be observed with NFS and with disk quota.

A successful close does not guarantee that the data has been successfully saved to disk, as the kernel defers writes. It is not common for a filesystem to flush the buffers when the stream is closed. If you need to be sure that the data is physically stored use **fsync**(2). (It will depend on the disk hardware at this point.)

It is probably unwise to close file descriptors while they may be in use by system calls in other threads in the same process. Since a file descriptor may be re-used, there are some obscure race conditions that may cause unintended side effects.

When dealing with sockets, you have to be sure that there is no **recv**(2) still blocking on it on another thread, otherwise it might block forever, since no more messages will be sent via the socket. Be sure to use **shutdown**(2) to shut down all parts the connection before closing the socket.

SEE ALSO

fcntl(2), **fsync**(2), **open**(2), **shutdown**(2), **unlink**(2), **fclose**(3)

NAME

read − read from a file descriptor

SYNOPSIS

#include <unistd.h>

ssize_t read(int *fd***, void ****buf***, size_t** *count***);**

DESCRIPTION

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

If *count* is zero, **read**() returns zero and has no other results. If *count* is greater than SSIZE_MAX, the result is unspecified.

RETURN VALUE

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because **read**() was interrupted by a signal. On error, −1 is returned, and *errno* is set appropriately. In this case it is left unspecified whether the file position (if any) changes.

ERRORS

EAGAIN
Non-blocking I/O has been selected using **O_NONBLOCK** and no data was immediately available for reading.

EBADF
fd is not a valid file descriptor or is not open for reading.

EFAULT
buf is outside your accessible address space.

EINTR
The call was interrupted by a signal before any data was read.

EINVAL
fd is attached to an object which is unsuitable for reading; or the file was opened with the **O_DIRECT** flag, and either the address specified in *buf*, the value specified in *count*, or the current file offset is not suitably aligned.

EIO I/O error. This will happen for example when the process is in a background process group, tries to read from its controlling tty, and either it is ignoring or blocking SIGTTIN or its process group is orphaned. It may also occur when there is a low-level I/O error while reading from a disk or tape.

EISDIR
fd refers to a directory.

Other errors may occur, depending on the object connected to *fd*. POSIX allows a **read**() that is interrupted after reading some data to return −1 (with *errno* set to EINTR) or to return the number of bytes already read.

CONFORMING TO

SVr4, 4.3BSD, POSIX.1-2001.

RESTRICTIONS

On NFS file systems, reading small amounts of data will only update the time stamp the first time, subsequent calls may not do so. This is caused by client side attribute caching, because most if not all NFS clients leave st_atime (last file access time) updates to the server and client side reads satisfied from the client's cache will not cause st_atime updates on the server as there are no server side reads. UNIX semantics can be obtained by disabling client side attribute caching, but in most situations this will substantially increase server load and decrease performance.

Many filesystems and disks were considered to be fast enough that the implementation of **O_NONBLOCK** was deemed unnecessary. So, O_NONBLOCK may not be available on files and/or disks.

SEE ALSO

close(2), **fcntl**(2), **ioctl**(2), **lseek**(2), **open**(2), **pread**(2), **readdir**(2), **readlink**(2), **readv**(2), **select**(2), **write**(2), **fread**(3)

NAME

write − write to a file descriptor

SYNOPSIS

#include <unistd.h>

ssize_t write(int *fd***, const void ****buf***, size_t** *count***);**

DESCRIPTION

write() writes up to *count* bytes to the file referenced by the file descriptor *fd* from the buffer starting at *buf*. POSIX requires that a **read**() which can be proved to occur after a **write**() has returned returns the new data. Note that not all file systems are POSIX conforming.

RETURN VALUE

On success, the number of bytes written are returned (zero indicates nothing was written). On error, −1 is returned, and *errno* is set appropriately. If *count* is zero and the file descriptor refers to a regular file, 0 may be returned, or an error could be detected. For a special file, the results are not portable.

ERRORS

EAGAIN
Non-blocking I/O has been selected using **O_NONBLOCK** and the write would block.

EBADF
fd is not a valid file descriptor or is not open for writing.

EFAULT
buf is outside your accessible address space.

EFBIG
An attempt was made to write a file that exceeds the implementation-defined maximum file size or the process' file size limit, or to write at a position past the maximum allowed offset.

EINTR
The call was interrupted by a signal before any data was written.

EINVAL
fd is attached to an object which is unsuitable for writing; or the file was opened with the **O_DIRECT** flag, and either the address specified in *buf*, the value specified in *count*, or the current file offset is not suitably aligned.

EIO A low-level I/O error occurred while modifying the inode.

ENOSPC
The device containing the file referred to by *fd* has no room for the data.

EPIPE *fd* is connected to a pipe or socket whose reading end is closed. When this happens the writing process will also receive a **SIGPIPE** signal. (Thus, the write return value is seen only if the program catches, blocks or ignores this signal.)

Other errors may occur, depending on the object connected to *fd*.

CONFORMING TO

SVr4, 4.3BSD, POSIX.1-2001.

Under SVr4 a write may be interrupted and return EINTR at any point, not just before any data is written.

NOTES

A successful return from **write**() does not make any guarantee that data has been committed to disk. In fact, on some buggy implementations, it does not even guarantee that space has successfully been reserved for the data. The only way to be sure is to call **fsync**(2) after you are done writing all your data.

SEE ALSO

close(2), **fcntl**(2), **fsync**(2), **ioctl**(2), **lseek**(2), **open**(2), **pwrite**(2), **read**(2), **select**(2), **writev**(3), **fwrite**(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 *filedes***, struct stat *** *buf* **);**
int lstat(const char * *path***, struct stat *** *buf* **);**

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 *filedes*.

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 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, for example, 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 filesystems 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, e.g. 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, e.g. 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.)

The following flags are defined for the *st_mode* field:

S_IFMT	0170000	bitmask for the file type bitfields
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
S_ISUID	0004000	set UID 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.

LINUX NOTES

Since kernel 2.5.48, the *stat* structure supports nanosecond resolution for the three file timestamp fields. Glibc exposes the nanosecond component of each field using names either of the form *st_atim.tv_nsec*, if the _BSD_SOURCE or _SVID_SOURCE feature test macro is defined, or of the form *st_atimensec*, if neither of these macros is defined. On file systems that do not support sub-second timestamps, these nanosecond fields are returned with the value 0.

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.

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**(2).)

EBADF
filedes 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.

CONFORMING TO

These system calls conform to SVr4, 4.3BSD, POSIX.1-2001.

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.)

POSIX does not describe the S_IFMT, S_IFSOCK, S_IFLNK, S_IFREG, S_IFBLK, S_IFDIR, S_IFCHR, S_IFIFO, S_ISVTX bits, but instead demands the use of the macros S_ISDIR(), etc. 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 userspace)
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 non-directories: 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	SysV 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.

SEE ALSO

access(2), **chmod**(2), **chown**(2), **fstatat**(2), **readlink**(2), **utime**(2), **capabilities**(7)