ECS 36A, May 21, 2025

Get File Status

```
int stat(const char *name, struct stat *buf)
int fstat(int fd, struct stat *buf)
```

- Get the status of the file *name* or the file associated with the descriptor *fd*

```
int lstat(const char *name, struct stat *buf)
```

- Get the status of the symbolic link rather than the file that the symbolic link points to

Representation of Status

```
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 file system I/O */
       blkcnt_t st_blocks;        /* number of 512 byte 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 */
};
```

Representation of Status

```
struct stat {
        dev_t st_dev; /* ID of device containing file */
        ino_t st_ino; /* inode number */
```

These two enable Linux to find the file: it goes first to the device with the device ID, and then looks for the inode, which contains information about where the fie is located as well as status information

```
    mode_t st_mode;             /* protection */
```

This contains the user, group, and other rights as a set of 9 bits, as well as 3 bits indicating privilege

```
    nlink_t st_nlink;           /* number of hard links */
```

A *hard link* is an alternate name for the file; a soft link is a file that contains the name of the file with which it is linked. Linux keeps track of the number of hard links, but not soft links. Example: for a directory, the number of hard links is at least 2 (the link from the parent directory to it, and the link from "." to it)

Representation of Status

```
uid_t st_uid;               /* user ID of owner */
gid_t st_gid;               /* group ID of owner */
```

These are the user id (of the owner) and the group id of the file. A file can be in at most 1 group.

```
dev_t st_rdev;              /* device ID (if special file) */
```

This is used for device files; they have a major and a minor number associated with them. Linux uses these to find the device driver (major number; it controls how the device does I/O), and passes the minor number to that driver.

```
off_t st_size;              /* total size, in bytes */
blksize_t st_blksize;       /* blocksize for file system I/O */
blkcnt_t st_blocks;         /* number of 512 byte blocks allocated */
```

These contain information about the size of the file. The first is the number of bytes in the file. The third contains the number of disk blocks that the file is using. The second is the number of bytes in each block on the device; this is useful for buffering.

Representation of Status

```
    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 */
```

These are the times of last file access, last modification (write), and last status change (when the file's metadata was changed; for example, changing protection mode).

```
};
```

Example: *filestat.c*

- Goal is to list all attributes of files
- Linux (and some other systems) have *extended attributes*
  - Two types: system attributes (set by system), user attributes (set by users)
  - These vary from system to system
  - So does how you access them (the MacOS system call has 2 more arguments than the Linux system call has, and uses a different system call for symbolic links)
- Not discussed here

Example: *filestat.c*

- These are in the routine *do_stat*, which gets the information and prints it in an intelligible format. *do_stat* is called from *main* with 1 file name
- This uses *lstat* as we want information about the named file, even if it is a symbolic link

```
/* get the file information and complain on error */
if (lstat(fname, &stbuf) < 0){
    perror(fname);
    return(0);
}

/* print file name */
printf("%s:\n", fname);
```

The information goes into a buffer, declared as `struct stat stbuf`

We need to take `stbuf`'s address as it is declared as a variable and the file status data needs to be placed there

Example: *filestat.c*

This prints basic information about the file that is related to the file system

```
/* print file system information */
printf("\t* File system information:\n");
printf("\t  Major ID of file's device: %d\n", (int) major(stbuf.st_dev));
printf("\t  Minor ID of file's device: %d\n", (int) minor(stbuf.st_dev));
printf("\t  Optimal file system blocksize: %ld\n", (long) stbuf.st_blksize);
printf("\t  File inode number %ld\n", (long) stbuf.st_ino);
```

The major, minor IDs and the inode number are used to locate the file metadata and contents

Optimal blocksize is the block size of the file system containing the file

Example: *filestat.c*

Print information about device files

- Character special devices are character-oriented, like terminals
- Block special devices are block-oriented, like hard drives

File type encoded in st_mode field

```
if (S_ISBLK(stbuf.st_mode) || S_ISCHR(stbuf.st_mode)){
        printf("\t  File is a device interface:\n");
        printf("\t\tMajor device number (driver number): %d\n",
                                          (int) major(stbuf.st_rdev));
        printf("\t\tMinor device number (driver dependent): %d\n",
                                          (int) minor(stbuf.st_rdev));
}
```

True if file is block special

True if file is character special

Example: *filestat.c*

This prints the type of file (function is *ftype(mode_t fperms))

```
if (S_ISREG(f_perms)) return("regular file");
else if (S_ISDIR(f_perms)) return("directory");
else if (S_ISCHR(f_perms)) return("character special device");
else if (S_ISBLK(f_perms)) return("block special device");
else if (S_ISFIFO(f_perms)) return("fifo (named pipe)");
else if (S_ISLNK(f_perms)) return("symbolic link");
else if (S_ISSOCK(f_perms)) return("socket");
/* should never get here, but just in case ... */
sprintf(perm_buf, "*** unknown type (%0o) ***", (f_perms>>12)&0xf);
return(perm_buf);
}
```

st_mode encodes file type; here we use macros to test for the file type

Example of defensive programming

Example: *filestat.c*

Now we handle symbolic links by printing the name of the file they are linked to. Note the link contains the name but that is *not* terminated by a NUL ('\0') byte.

```
if (S_ISLNK(stbuf.st_mode)){
        printf(" [");
        if ((lname = malloc(stbuf.st_size+1)) == NULL)
                perror("malloc");
        else if ((r = readlink(fname, lname, stbuf.st_size + 1)) < 0)
                perror("readlink");
        else if (r > stbuf.st_size)
                fprintf(stderr, "*** Link contents changed during read ***");
        else{
                lname[stbuf.st_size] = '\0';
                printf("%s", lname);
        }
        printf("]");
}
```

st_mode encodes file type; here we use a macro to see if it is a symbolic link

Allocate space; all the link file contains is the name of another file, so it's file size plus 1.

Read the link

Check for an unexpected change. If the file name is different than the size of the link obtained earlier, then the link was changed and the space allocated for the name is now be too small.

Read the link

Example: *filestat.c*

- The number of links is, essentially, the number of names for the file.
- The sizes are self-explanatory. The reason for the 512 bytes for the bocks is because some file systems can split disk blocks into fragments of 512 bytes.

```
printf("\t  Number of hard links: %ld\n", (long) stbuf.st_nlink);
printf("\t  Number of bytes in file: %ld\n", (long) stbuf.st_size);
printf("\t  Number of 512 byte blocks allocated: %ld\n", (long)
                                                    stbuf.st_blocks);
```

Example: *filestat.c*

These are the bits for permissions. There are 12 of them

- The first 3 are privilege bits
- The next 3 are the user rights (read, write, execute)
- The next 3 are the group rights (read, write, execute)
- The next 3 are the other rights (read, write, execute)

```
/* how many bits to shift left to get the set needed */
#define PRIV 9 /* privilege bits */
#define USER 6 /* permissions for UID */
#define GROUP 3 /* permissions for GID */
#define OTHER 0 /* permissions for everyone else */
char *p_rights[] = { "setuid", "setgid", "sticky" };
char *f_rights[] = { "read", "write", "execute" };
char *d_rights[] = { "list", "modify", "search" };
char perm_buf[128]; /* big enough to hold response */
```

These are the shifts needed to make the corresponding bits be the low ones in the word; we can then and them with 7 to get them

privileges corresponding to the privilege bits

permissions corresponding to the permission bits for files bits for files

permissions corresponding to the permission bits for files bits for directories

small buffer to hold permission, privilege names

Example: *filestat.c*

```
if (S_ISDIR(f_perms))
        rights = d_rights;
else
        rights = f_rights;

switch(what){
case USER: case GROUP: case OTHER:
        bits = (f_perms>>what)&07;
        break;
default:
        return("*** internal error ***");
}
perm_buf[0] = '\0';
```

The parameters are:

- `what`, which says which set of rights (USER, GROUP, OTHER) are to be printed
- `f_perms`, which is the set of permission bits

This tests whether it's a directory, and sets the rights names appropriately. `d_rights` give the names of directory rights, and `f_rights` the names of file rights.

Now we get the set of bits for the proper set of permissions; it's shifted by the appropriate number of bits and the low-order 3 bits are extracted

"Can't happen" so we check for it (remember,

Example: *filestat.c*

```
switch(bits){
case 7: /* read, write, and execute / list, modify, and search */
        (void) strcpy(perm_buf, rights[0]);
        (void) strcat(perm_buf, ", ");
        (void) strcat(perm_buf, rights[1]);
        (void) strcat(perm_buf, ", and ");
        (void) strcat(perm_buf, rights[2]);
        break;
case 6: /* read and write */
        (void) strcpy(perm_buf, rights[0]);
        (void) strcat(perm_buf, " and ");
        (void) strcat(perm_buf, rights[1]);
        break;
```

Proceed in the obvious way: build `perm_buf`'s contents to list the rights. There are 8 of these, from 0 to 7. These are the first two.

Example: *filestat.c*

- Now the time. These format the time and put it into an array.

```
char p_time[1024];
```

The time is placed in this array

```
char *timefmt = "%A, %B %e, %Y at %l:%M:%S%p %Z";
```

- Time format string; the result looks like this:

%A = full name of day of week	%Y = full year	%S = seconds (0-59)
%B = full name of month	%l = hour of day (1-12)	%p = AM or PM
%e = day of month (1-31)	%M = minutes (0-59)	%Z =time zone

- So this comes out as:

```
Wednesday, May 21, 2025 at  5:27:32AM PDT
```

Example: *filestat.c*

- This is the function that takes the internal representation of time (*tock*) and converts it into the string on the previous slide

```
char *prtime(time_t tock)
{
        struct tm ticktock;

        ticktock = *localtime(&tock);
        if (!strftime(p_time, sizeof(p_time), timefmt, &ticktock))
                sprintf(p_time, "%ld", tock);
        return(p_time);
}
```

Convert internal time representation to a time structure

Now convert it into a string formatted as the `timefmt` requires, and store it in `p_time`

If the conversion fails, return the internal time representation

Closing a File

```
int close(int fd)
```

- Disassociates the file associated with *fd*
- *fd* no longer is bound to any file and can be reused
- On success, it returns 0
- On failure, it returns –1 and puts the error code in *errno*

Deleting a File

```
int unlink(const char *name)
```

- Deletes file *name* from the file system
- If there are other links to it, the file's storage is still being used
  - If the file is open, that's a link
- On success, it returns 0
- On failure, it returns –1 and puts the error code in *errno*

Example: Deleting Files

```
$ touch xyzzy
$ ln xyzzy plugh
$ ls -il
total 36
4134245867 -rwxr-xr-x 1 bishop bishop 21288 May 23 23:15 filestat
4035671211 -rw-r--r-- 1 bishop bishop  7365 May 23 23:15 filestat.c
4134245866 -rw------- 2 bishop bishop     0 May 23 23:22 plugh
4134245866 -rw------- 2 bishop bishop     0 May 23 23:22 xyzzy
```

Create a file (*touch* either creates the file or changes the modification time

Give it a second name (*ln* stands for "link")

The –i option lists the inode number; each file has a single inode number, so 1 file with 2 names will show both file names and they have the same inode

These two files have the same inode number and so are different names for the same file

Example: Deleting Files

```
$ rm xyzzy
rm: remove regular empty file 'xyzzy'? y
$ ls -il
total 36
4134245867 -rwxr-xr-x 1 bishop bishop 21288 May 23 23:15 filestat
4035671211 -rw-r--r-- 1 bishop bishop  7365 May 23 23:15 filestat.c
4134245866 -rw------- 1 bishop bishop     0 May 23 23:22 plugh
bishop@COE-CS-pc12:~/ecs36a/0521$
```

Now delete the original file

Verifying so I don't delete the wrong thing!

The –i option lists the inode number

The other name for the deleted file shows up, and it has the same inode number as before; so it is really the same file with a different name