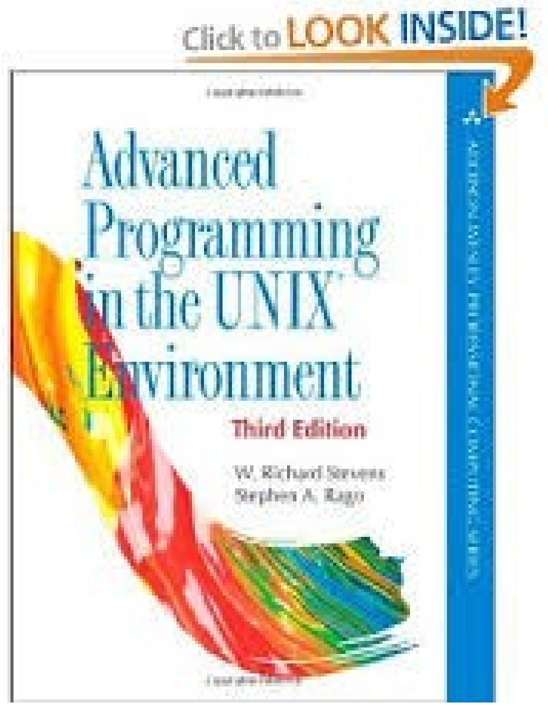
COP 4338 Class 11

Last Class:

-Systems programming!
-Unix system overview
-File I/O

Today's Class:

File I/O, stats, directories


Click to LOOK INSIDE!
Advanced Programming in the UNIX Environment
Third Edition
W. Richard Stevens
Stephen A. Rago

Chapter 3

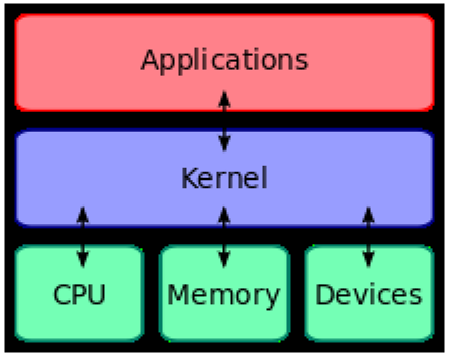
Where are we?

- -C fundamentals
  -Memory (Previous Classes)
- -Unix Fundamentals
- -Files (today)
- -Processes
- -Threads
- -Interprocess communication
- -Parallel Computing
- -Networking
- -Optional Advanced Topics
-

Review System Calls and Library Functions

- 1: User command; man-pages includes a very few Section 1 pages that document programs supplied by the GNU C library.
- 2: System calls documents the system calls provided by the Linux kernel.
- 3: Library functions documents the functions provided by the standard C library.

- man 2 intro
- man 2 syscalls
- ls /bin
-
-



Reminder:System Calls

- When a computer is turned on, the program that gets executed first is called the ``operating system." It controls pretty much all activity in the computer. This includes who logs in, how disks are used, how memory is used, how the CPU is used, and how you talk with other computers. The operating system we use is called "Unix".

- The way that programs talk to the operating system is via ``system calls." A system call looks like a procedure call but it's different -- it is a request to the operating system to perform some activity.

- System calls are expensive. While a procedure call can usually be performed in a few machine instructions, a system call requires the computer to save its state, let the operating system take control of the CPU, have the operating system perform some function, have the operating system save its state, and then have the operating system give control of the CPU back to you.

File I/O

We'll continue our discussion of the UNIX System by describing the functions available for file I/O—open a file,

The functions described in this class are often referred to as unbuffered I/O, in contrast to the standard I/O routines, which we describe last week. The term unbuffered means that each read or write invokes a system call in the kernel.

These unbuffered I/O functions are not part of ISO C, but are part of POSIX.1 and the Single UNIX Specification

File I/O

We'll continue our discussion of the UNIX System by describing the functions available for file I/O—open a file,

The functions described in this class are often referred to as unbuffered I/O, in contrast to the standard I/O routines, which we describe last week. The term unbuffered means that each read or write invokes a system call in the kernel.

These unbuffered I/O functions are not part of ISO C, but are part of POSIX.1 and the Single UNIX Specification

File Descriptors

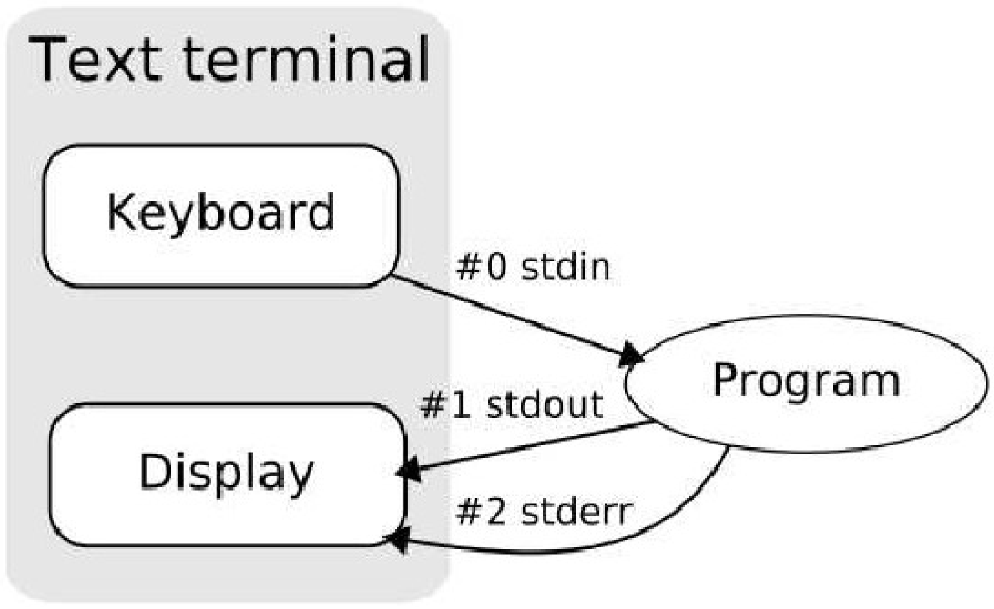
To the kernel, all open files are referred to by file descriptors.

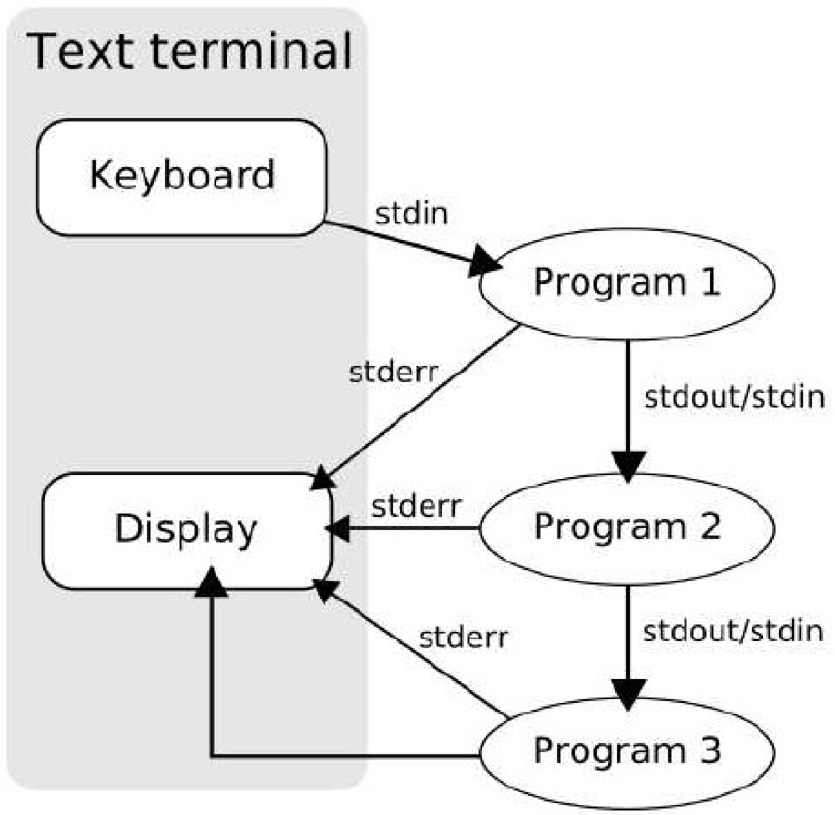
- A file descriptor (or file handle) is a small, non-negative integer which identifies a file to the kernel.
- Traditionally, stdin, stdout and stderr are 0, 1 and 2 respectively.
- Relying on “magic numbers” is Bad. Use STDIN_FILENO, STDOUT_FILENO and STDERR_FILENO.
- These constants are defined in the <unistd.h> header.

```
more /usr/include/unistd.h
```

-

File Descriptors





File Descriptors

File descriptors range from 0 through OPEN_MAX.

Run openmax.c

Read:

http://en.wikipedia.org/wiki/File_descriptor

Standard I/O

Reminder Basic File I/O: almost all UNIX file I/O can be performed using these five functions:

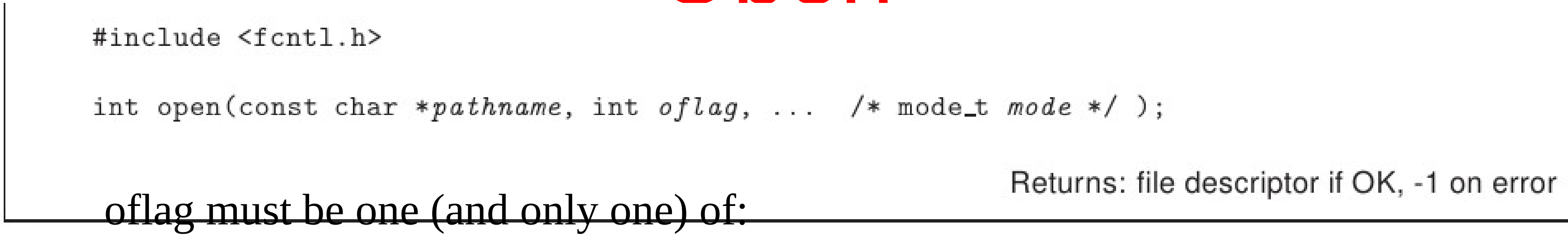
- open(2)
- close(2)
- lseek(2)
- read(2)
- write(2)
-

Standard I/O

Processes may want to share resources. This requires us to look at:

- atomicity of these operations
- file sharing
- manipulation of file descriptors

Open



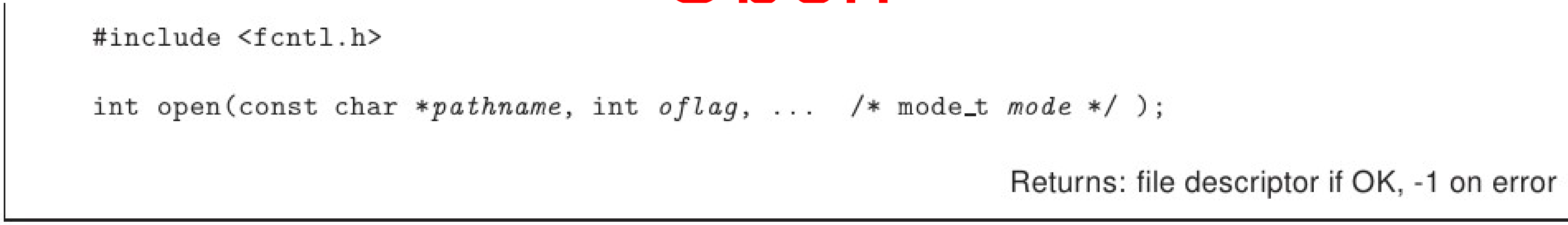
```
#include <fcntl.h>

int open(const char *pathname, int oflag, ...  /* mode_t mode */ );
```

Returns: file descriptor if OK, -1 on error

oflag must be one (and only one) of:
O_RDONLY – Open for reading only
O_WRONLY – Open for writing only
O_RDWR – Open for reading and writing

and may be OR'd with any of these:
O_APPEND – Append to end of file for each write
O_CREAT – Create the file if it doesn't exist.
O_EXCL – Generate error if O CREAT and file already exists. (atomic)
O_TRUNC – If file exists and successfully open in O WRONLY or O RDWR, make length = 0
O_NOCTTY – If pathname refers to a terminal device, do not allocate the device as a controlling terminal
O_NONBLOCK – If pathname refers to a FIFO, block special, or char special, set nonblocking mode (open and I/O)
O_SYNC – Each write waits for physical I/O to complete

Open



```
#include <fcntl.h>

int open(const char *pathname, int oflag, ...  /* mode_t mode */ );
```

Returns: file descriptor if OK, -1 on error

On some platforms oflag may also be one of:
O_EXEC – Open for execute only
O SEARCH – Open for search only (applies to directories)
and may be OR'd with any of these:
O DIRECTORY – If path resolves to a non-directory file, fail and set errno to ENOTDIR.
O DSYNC – Wait for physical I/O for data, except file attributes
O RSYNC – Block read operations on any pending writes.
O PATH – Obtain a file descriptor purely for fd-level operations. (Linux >2.6.36 only)

close(2)

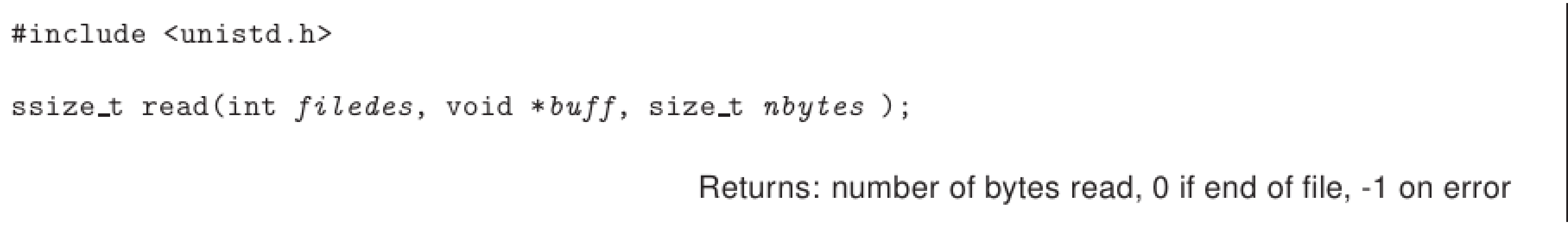
```
#include <unistd.h>

int close(int fd);

                                        Returns: 0 if OK, -1 on error
```

- closing a filedescriptor releases any record locks on that file (more on that in future lectures)
- file descriptors not explicitly closed are closed by the kernel when the process terminates.

Open(2) and close(2)

- open-ex.c
- c1.c

read(2)



```
#include <unistd.h>

ssize_t read(int filedes, void *buff, size_t nbytes );
```

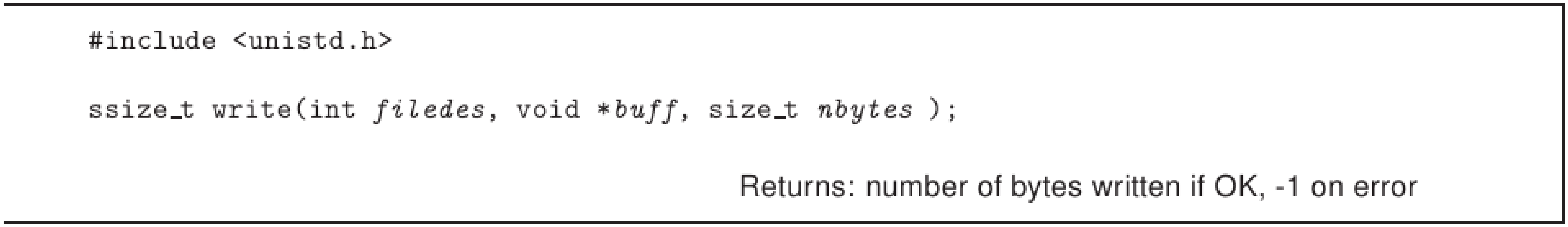
Returns: number of bytes read, 0 if end of file, -1 on error

There can be several cases where read returns less than the number of bytes requested:

- EOF reached before requested number of bytes have been read
- Reading from a terminal device, one ”line” read at a time
- Reading from a network, buffering can cause delays in arrival of data
- Interruption by a signal

read begins reading at the current offset, and increments the offset by the number of bytes actually read.

write(2)



```
#include <unistd.h>

ssize_t write(int filedes, void *buff, size_t nbytes );
```

Returns: number of bytes written if OK, -1 on error

write returns nbytes or an error has occurred (disk full, file size limit exceeded, …) for regular files, write begins writing at the current offset (unless O APPEND has been specified, in which case the offset is first set to the end of the file)

- after the write, the offset is adjusted by the number of bytes actually written

write(2)

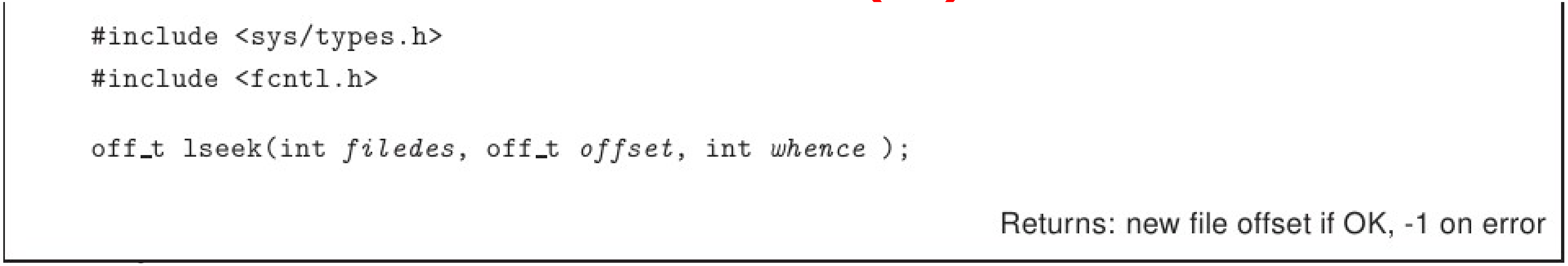
```
#include <unistd.h>

ssize_t write(int filedes, void *buff, size_t nbytes );
```

Returns: number of bytes written if OK, -1 on error

examplewrite1.c

- examplewrite2.c
- rwex.c
- w1.c

lseek(2)



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

off_t lseek(int filedes, off_t offset, int whence );
```

Returns: new file offset if OK, -1 on error

The value of whence determines how offset is used:
SEEK SET bytes from the beginning of the file
SEEK CUR bytes from the current file position
SEEK END bytes from the end of the file
“Weird” things you can do using lseek(2):
-seek to a negative offset
-seek 0 bytes from the current position
-seek past the end of the file
man fseek
man lseek

lseek(2)

Move to byte #16

```
newpos = lseek(fd, 16, SEEK_SET);
```

Move forward 4 bytes

```
newpos = lseek(fd, 4 , SEEK_CUR);
```

Move to 8bytes from the end

```
newpos = lseek(fd,8, SEEK_END);
```

lseek(2)

newpos = lseek(fd, 16, SEEK_SET);

-

newpos = lseek(fd, 4 , SEEK_CUR);

newpos = lseek(fd,8, SEEK_END);

lseek(2)

- Run l1.c

Standard Input, Standard Output, and Standard Error

Now, every process in Unix starts out with three file descriptors predefined and open:

- File descriptor 0 is standard input.
- File descriptor 1 is standard output.
- File descriptor 2 is standard error.

- Thus, when you write a program, you can read from standard input, using read(0, ...), and write to standard output using write(1, ...).

- Armed with that information, we can write a very simple cat program (one that copies standard input to standard output) with one line
-
- run simpcat.c

File I/O

- System Call
  - `open`
  - `close`
  - `read/write`
  - `lseek`

- Standard I/O call (`stdio.h`)
  - `fopen`
  - `fclose`
  - `getchar/putchar, getc/putc, fgetc/fputc, fread/fwrite, gets/puts, fgets/fputs, scanf/printf, fscanf/fprintf`
  - `fseek`

Files

Files store information on secondary storage. This means that the information should exist even when the computer that stored it is powered off. This is as opposed to primary storage, which only works while a computer is powered up, and which goes away forever when a computer is powered down.

- When you create a file in Unix, there are quite a few things that happen. In this lecture, we are going to focus on three components of a file in Unix:

- The bytes of the file itself.
- The metadata of the file.
- The links to the file that are relative to a directory.

```
> echo "This is f1.txt" > f1.txt
```

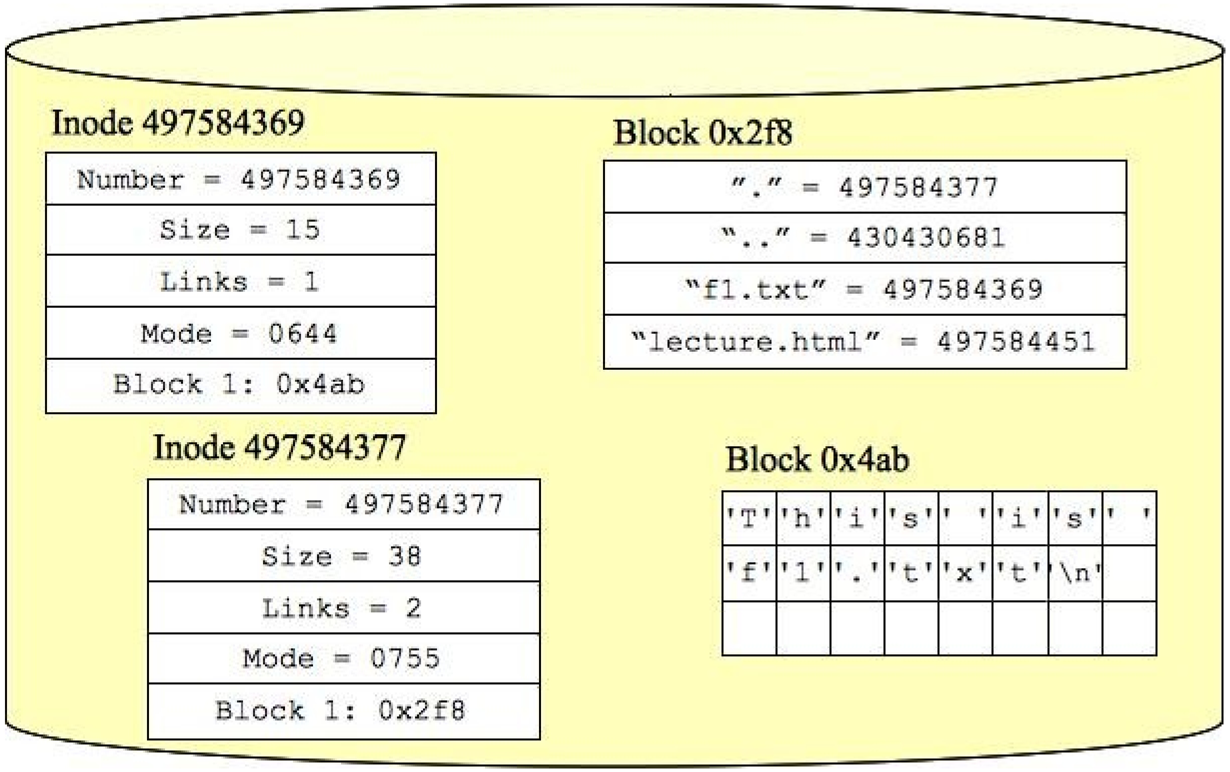
- `>ls -lai`

Files

-3 Things happen when you create a directory

- A directory entry for "f1.txt" in the current directory. That associates the name "f1.txt" with an "inode" with a number
- An "Inode" with a number. This contains metadata, or information about the file. Typical metadata is the size, owner, links, protection mode, latest access and modification time, etc. This is stored on disk in its own location -- the operating system knows how to find it by its inode number.
- The actual bytes, "This is f1.txt". These are placed in a disk block, and the inode has information on how to find the disk block. In this example, the file is contained within one disk block, and the inode will know how to find it. i

Files



Inode 497584369

Number = 497584369
Size = 15
Links = 1
Mode = 0644
Block 1: 0x4ab

Block 0x2f8

"." = 497584377
".." = 430430681
"f1.txt" = 497584369
"lecture.html" = 497584451

Inode 497584377

Number = 497584377
Size = 38
Links = 2
Mode = 0755
Block 1: 0x2f8

Block 0x4ab

'T'	'h'	'i'	's'	' '	'i'	's'	' '
'f'	'1'	'.'	't'	'x'	't'	'\n'	

Files

To use Unix lingo, the way we name a file is by attaching a "link" to the inode. Links are stored in "directories" -- each entry in a directory maps the name of the link to the inode number of the inode that points to the file.

Files

```
>echo "This is f1.txt" > f1
>ln f1 f2
> ls -li f1 f2
>vi f2
>cat f2
```

Files

```
> chmod 0400 f1
>ln f1 f3
> ls -li f1 f2
> ls -li f1 f2 f3
> chmod 0644 f1
>rm f1
```

When the last link to a file is removed, then the file itself, inode and all, is deleted. As long as there is a link pointing to a file, however, the file remains

Files

All directories have at least 2 links:

- UNIX> mkdir test

  UNIX> ls -li | grep test

  34800 drwxr-xr-x 2 jabobadi   512 Sep 16 10:17 test
- UNIX>

- This is because every directory contains two subdirectories "." and ".." The first is a link to itself, and the second is a link to the parent directory. Thus, there are two links to the directory file "test": "test" and "test/." Similarly, suppose we make a subdirectory of test:

- UNIX> mkdir test/sub
- UNIX> ls -li | grep test
- 34800 drwxr-xr-x 3 jabobadi   512 Sep 16 10:17 test

- Now there are three links to "test": "test", "test/.", and "test/sub/.."

Soft-links

Besides these links which are automatically created for you, you cannot manually create links to directories. Instead, there is a special kind of a link called a "soft link", which you make using the command "ln -s". For example, we can create a soft link to the test directory as follows:

```
UNIX> ln -s test test-soft
UNIX> ls -li | grep test
34800  drwxr-xr-x  3 jabobadi        512 Sep 16 10:17 test
34801  lrwxrwxrwx  1 jabobadi          4 Sep 16 10:18 test-soft -> test
```

- Note that soft links have a different kind of directory listing. Moreover, note that the creation of a soft link to "test" doesn't update the link field of test's inode. That only records regular, or "hard" links.

Soft-links

A soft link is a way of pointing to a file without changing the file's inode. However, soft links can do pretty much everything that hard links can do:

```
UNIX> cat > f1
This is f1
UNIX> ln -s f1 f2
UNIX> cat f2
This is f1
UNIX> cat > f2
This is f2
UNIX> cat f1
This is f2
UNIX> ls -l f*
-rw-r--r--  1 plank          11 Sep 16 10:19 f1
lrwxrwxrwx  1 plank           2 Sep 16 10:18 f2 -> f1
UNIX> chmod 0600 f2
UNIX> ls -l f*
-rw-------  1 plank          11 Sep 16 10:19 f1
lrwxrwxrwx  1 plank           2 Sep 16 10:18 f2 -> f1
UNIX>
```

Soft-links

What is the main difference between hard and soft links then? Well, for one, if you delete all the hard links to a file, but not all the soft links, then the file still gets deleted.

```
UNIX> rm f1
UNIX> ls -l f*
lrwxrwxrwx  1 plank           2 Sep 16 10:18 f2 -> f1
UNIX> cat f2
cat: f2: No such file or directory
UNIX>
```

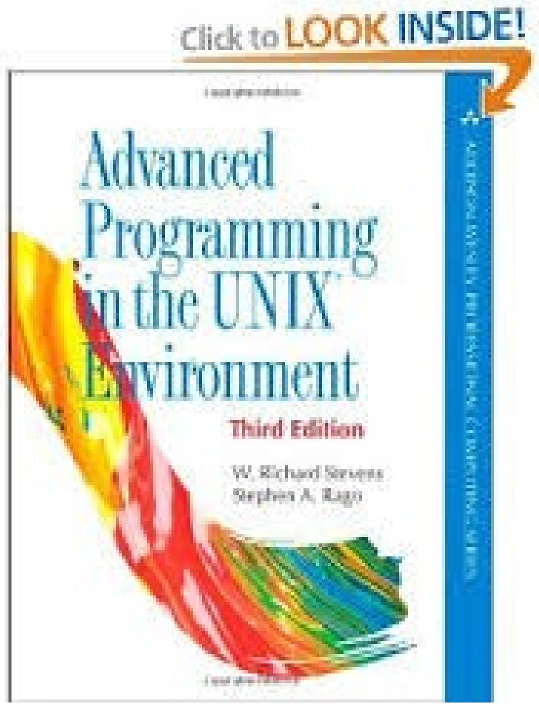
- The link is called "unresolved."

- In unix, you cannot make hard links from a file in one filesystem to a directory in another filesystem. However, you can make a soft link: ml

Try this at home

```
$ mkdir exampledir
$ cd exampledir
$ ls -n
$ touch f1
$ touch f2
$ ln f1 hardlink
$ ln -s f2 softlink
$ mkdir d1
$ mkdir d2
$ mkdir d2/a d2/b d2/c
$ ls -n
```

For a directory, the number of hard links is the number of immediate subdirectories it has plus its parent directory and itself. More details later



Chapter 4: Chapter 4. Files and Directories

File and Directories

We will cover stat, lstat, and the opendir

stat(2) family of functions

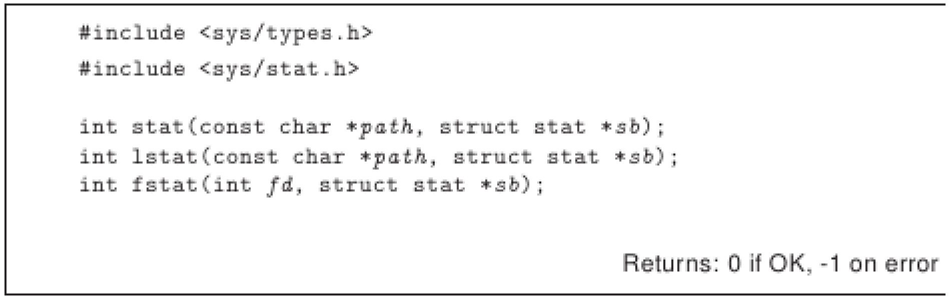
compile ls1.c

more /usr/include/sys/stat.h

man 2 stat

compile ls1.c

stat(2) family of functions



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

int stat(const char *path, struct stat *sb);
int lstat(const char *path, struct stat *sb);
int fstat(int fd, struct stat *sb);
```

Returns: 0 if OK, -1 on error

All these functions return extended attributes about the referenced file (in the case of symbolic links, lstat(2) returns attributes of the link, others return stats of the referenced file).

stat(2) family of functions

```
struct stat {
    dev_t      st_dev;          /* device number (filesystem) */
    ino_t      st_ino;          /* i-node number (serial number) */
    mode_t     st_mode;         /* file type & mode (permissions) */
    dev_t      st_rdev;         /* device number for special files */
    nlink_t    st_nlink;        /* number of links */
    uid_t      st_uid;          /* user ID of owner */
    gid_t      st_gid;          /* group ID of owner */
    off_t      st_size;         /* size in bytes, for regular files */
    time_t     st_atime;        /* time of last access */
    time_t     st_mtime;        /* time of last modification */
    time_t     st_ctime;        /* time of last file status change */
    long       st_blocks;       /* number of 512-byte* blocks allocated */
    long       st_blksize;      /* best I/O block size */
};
```

stat(2) family of functions

compile ls2.c

directories

Next, we'd like to have our "ls" work like the real "ls" -- have it accept no arguments, and list all files in the current directory.

directories

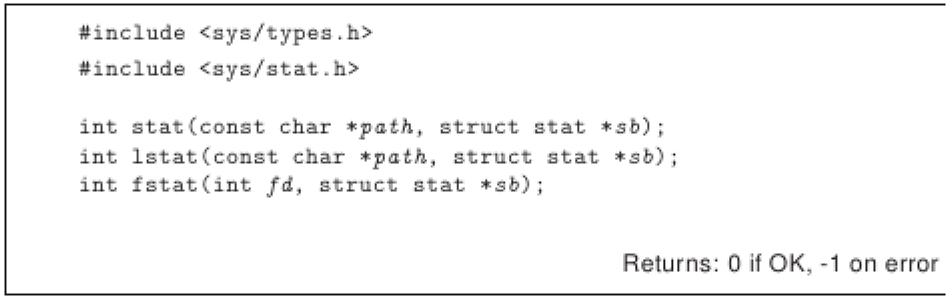
To do this, we need to use the "opendir/readdir/writedir" calls. Read the man page. Note that these are C library calls, and not system calls. This means that they call open/close/read/write, and interpret the format of directory files for you. This is convenient. The "struct dirent" structure is defined in /usr/include/sys/dirent.h:

```
struct  dirent {
        off_t           d_off;          /* offset of next disk dir entry */
        unsigned long   d_fileno;       /* file number of entry */
        unsigned short  d_reclen;       /* length of this record */
        char            *d_name;        /* name */
};
```

directories

compile ls3.c

stat(2) family of functions



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

int stat(const char *path, struct stat *sb);
int lstat(const char *path, struct stat *sb);
int fstat(int fd, struct stat *sb);
```

Returns: 0 if OK, -1 on error

All these functions return extended attributes about the referenced file (in the case of symbolic links, lstat(2) returns attributes of the link, others return stats of the referenced file).

stat(2) family of functions

```
struct stat {
    dev_t     st_dev;        /* device number (filesystem) */
    ino_t     st_ino;        /* i-node number (serial number) */
    mode_t    st_mode;       /* file type & mode (permissions) */
    dev_t     st_rdev;       /* device number for special files */
    nlink_t   st_nlink;      /* number of links */
    uid_t     st_uid;        /* user ID of owner */
    gid_t     st_gid;        /* group ID of owner */
    off_t     st_size;       /* size in bytes, for regular files */
    time_t    st_atime;      /* time of last access */
    time_t    st_mtime;      /* time of last modification */
    time_t    st_ctime;      /* time of last file status change */
    long      st_blocks;     /* number of 512-byte* blocks allocated */
    long      st_blksize;    /* best I/O block size */
};
```

struct stats

statexample.c

stat(2) st_mode

The st_mode field of the struct stat encodes the type of file:

- regular – most common, interpretation of data is up to application
- directory – contains names of other files and pointer to information on those files. Any process can read, only kernel can write.
- character special – used for certain types of devices
- block special – used for disk devices (typically). All devices are either character or block special.

- FIFO – used for interprocess communication (sometimes called named pipe)
- socket – used for network communication and non-network communication (same host).
- symbolic link – Points to another file.

  Find out more in <sys/stat.h>

-

struct stat: st mode, st uid and st gid

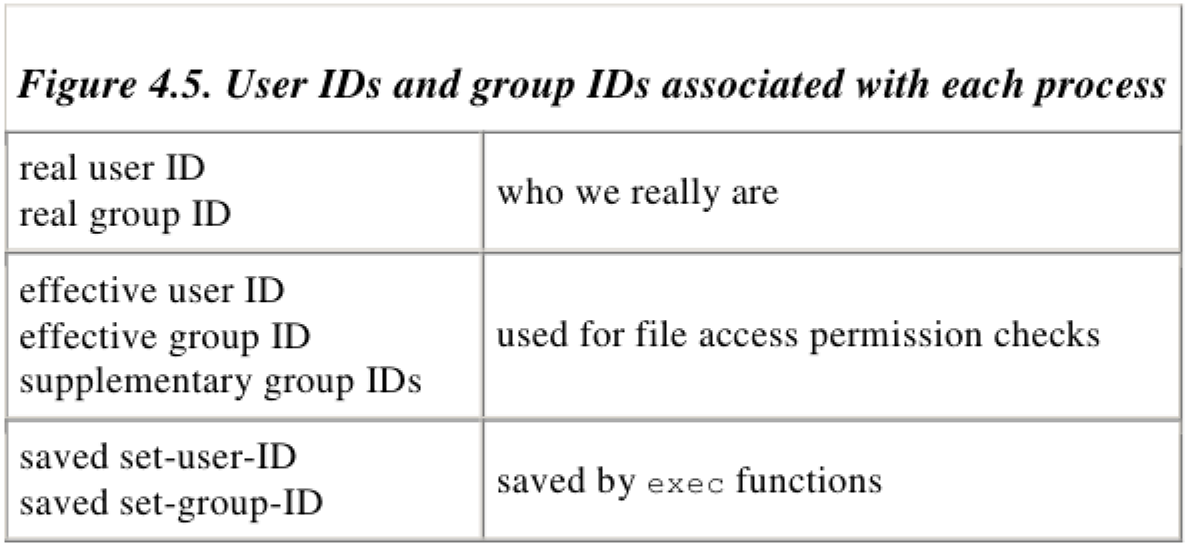


Figure 4.5. User IDs and group IDs associated with each process

real user ID real group ID	who we really are
effective user ID effective group ID supplementary group IDs	used for file access permission checks
saved set-user-ID saved set-group-ID	saved by `exec` functions

The real user ID and real group ID identify who we really are. These two fields are taken from our entry in the password file when we log in. Normally, these values don't change during a login session.
Normally, the effective user ID equals the real user ID, and the effective group ID equals the real group ID.

stat(2) family of functions

Every file has an owner and a group owner. The owner is specified by the st_uid member of the stat structure; the group owner, by the st_gid member.

-
- ls -l

File Access Permissions

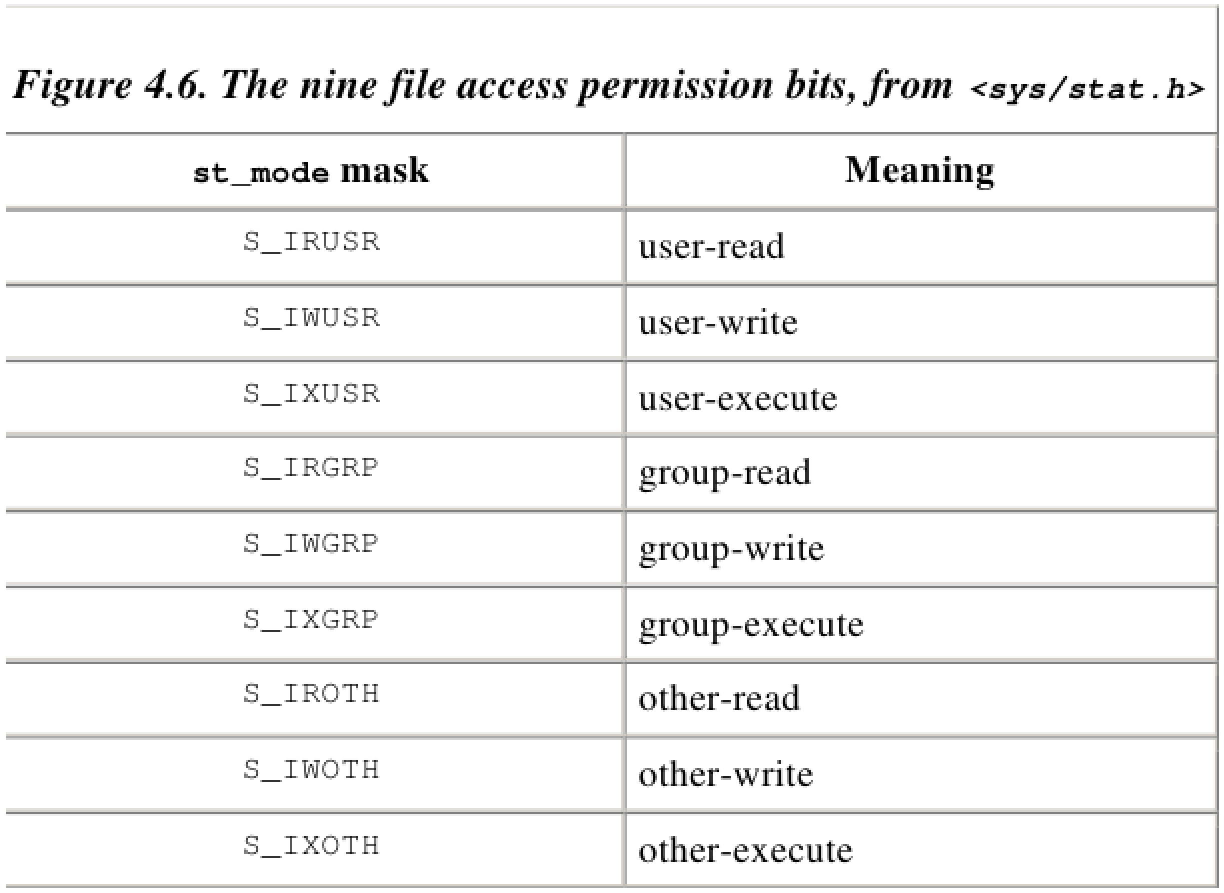


Figure 4.6. The nine file access permission bits, from `<sys/stat.h>`

`st_mode` mask	Meaning
`S_IRUSR`	user-read
`S_IWUSR`	user-write
`S_IXUSR`	user-execute
`S_IRGRP`	group-read
`S_IWGRP`	group-write
`S_IXGRP`	group-execute
`S_IROTH`	other-read
`S_IWOTH`	other-write
`S_IXOTH`	other-execute

File Access Permissions

st_mode also encodes the file access permissions (S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S IXGRP, S IROTH, S IWOTH, S IXOTH). Uses of the permissions are summarized as follows:

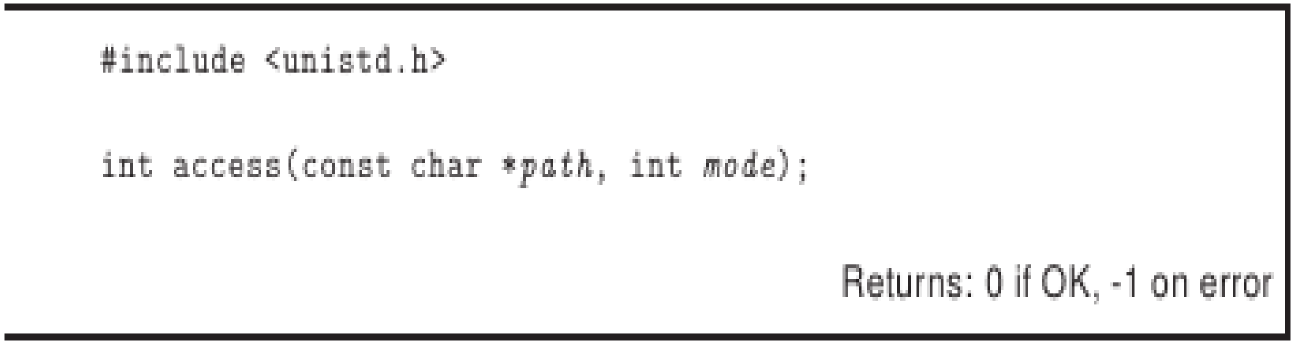
- To open a file, need execute permission on each directory component of the path
- To open a file with O_RDONLY or O_RDWR, need read permission
- To open a file with O_WRONLY or O_RDWR, need write permission
- To use O_TRUNC, must have write permission
- To create a new file, must have write+execute permission for the directory
- To delete a file, need write+execute on directory, file doesn’t matter
- To execute a file need execute permission

File Access Permissions

For example, to open the file /usr/include/stdio.h, we need execute permission in the directory /, execute permission in the directory /usr, and execute permission in the directory /usr/include. We then need appropriate permission for the file itself, depending on how we're trying to open it: read-only,read–write, and so on.

- For directories “execute” is the allowance to enter the directory using the cd command.

access(2)



```
#include <unistd.h>

int access(const char *path, int mode);
```

Returns: 0 if OK, -1 on error

Tests file accessibility on the basis of the real uid and gid. Allows setuid/setgid programs to see if the real user could access the file without it having to drop permissions to do so.

The mode paramenter can be a bitwise OR of:

- R_OK – test for read permission
- W_OK – test for write permission
- X_OK – test for execute permission
- F_OK – test for existence of file

access(2)

access.c

access(2)

access.c

```
sudo chown root a.out
sudo chmod 755 a.out
```

- http://en.wikipedia.org/wiki/Chmod

umask

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

mode_t umask(mode_t numask);
```

Returns: previous file mode creation mask

`umask(2)` sets the file creation mode mask. Any bits that are *on* in the file creation mask are turned *off* in the file's mode.

Important because a user can set a default umask. If a program needs to be able to insure certain permissions on a file, it may need to turn off (or modify) the umask, which affects only the current process.

umask

```
umask 022
touch foo
./umaskexample
```

umask

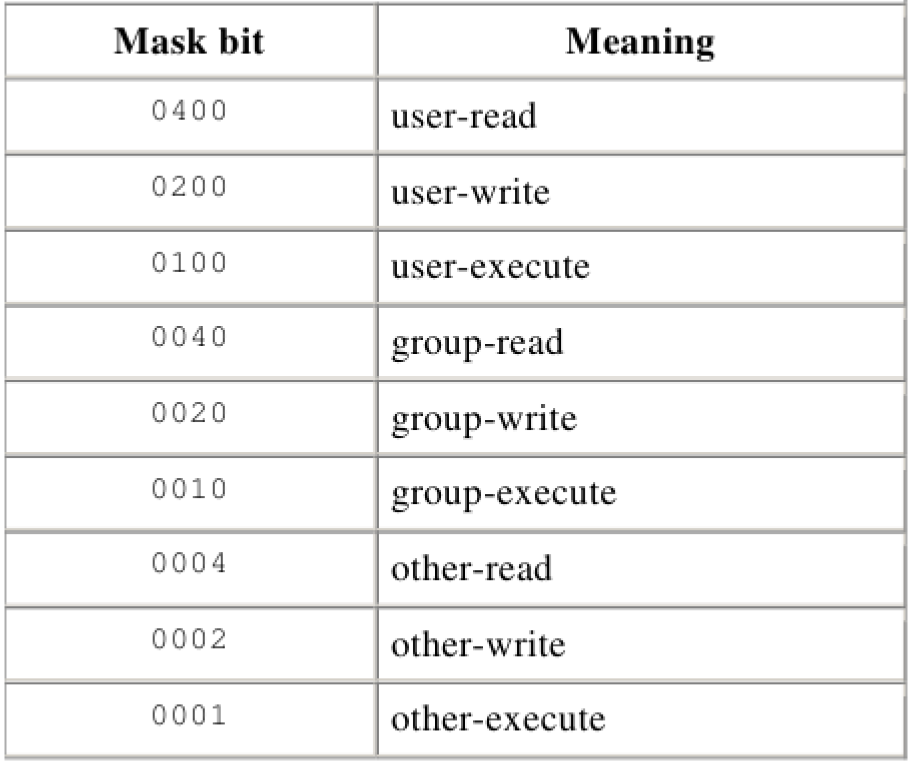


Figure 4.10. The `umask` *file access permission bits*

Mask bit	Meaning
`0400`	user-read
`0200`	user-write
`0100`	user-execute
`0040`	group-read
`0020`	group-write
`0010`	group-execute
`0004`	other-read
`0002`	other-write
`0001`	other-execute

umask

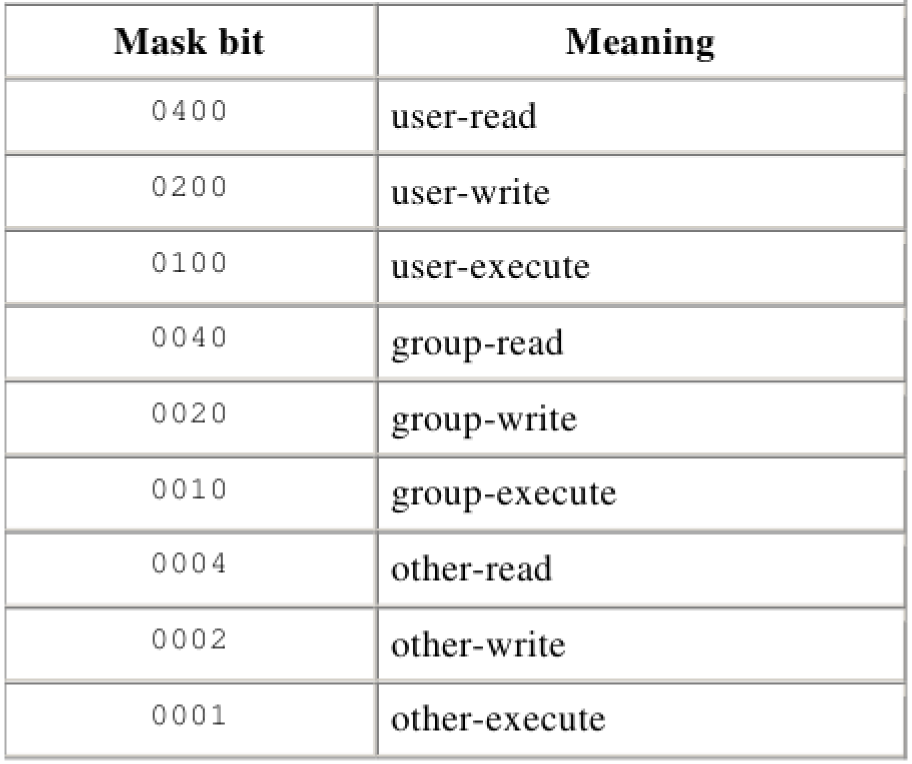


Figure 4.10. The `umask` file access permission bits

Mask bit	Meaning
0400	user-read
0200	user-write
0100	user-execute
0040	group-read
0020	group-write
0010	group-execute
0004	other-read
0002	other-write
0001	other-execute

chmod(2), lchmod(2) and fchmod(2)

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

int chmod(const char *path, mode_t mode);
int lchmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);

                                        Returns: 0 if OK, -1 on error
```

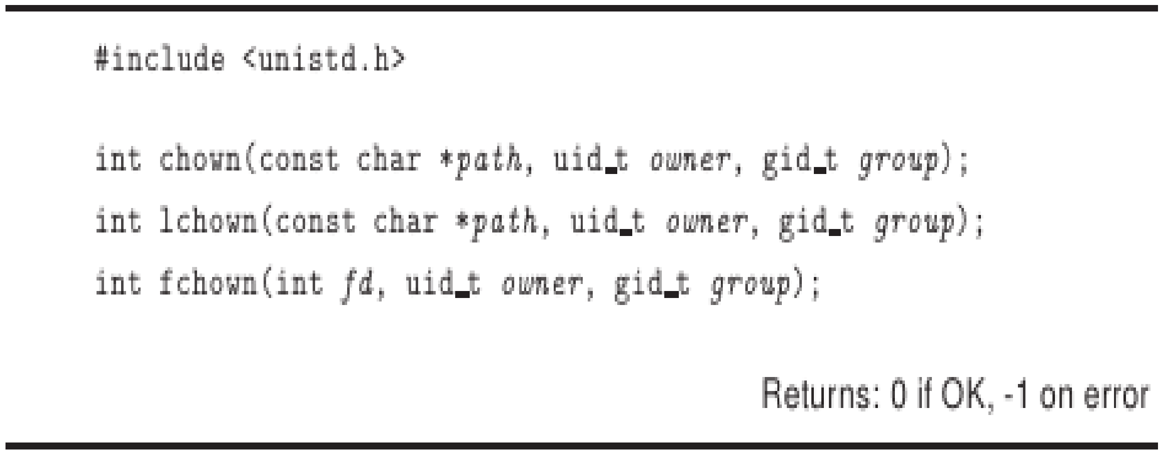
Changes the permission bits on the file. Must be either superuser or *effective uid* == `st_uid`. *mode* can be any of the bits from our discussion of `st_mode` as well as:

- `S_ISUID` – setuid
- `S_ISGID` – setgid
- `S_ISVTX` – sticky bit (aka “saved text”)
- `S_IRWXU` – user read, write and execute
- `S_IRWXG` – group read, write and execute
- `S_IRWXO` – other read, write and execute

chmod(2)

```
 rm foo*
 umask 077
 touch foo foo1
 chmod a+rx foo
 ls -l foo*
./chmodexample
```

chown(2)

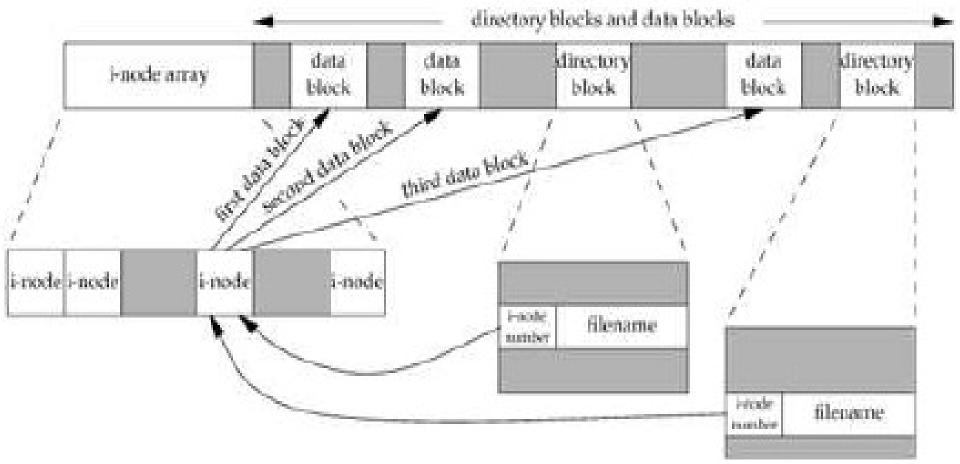


```
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

Returns: 0 if OK, -1 on error

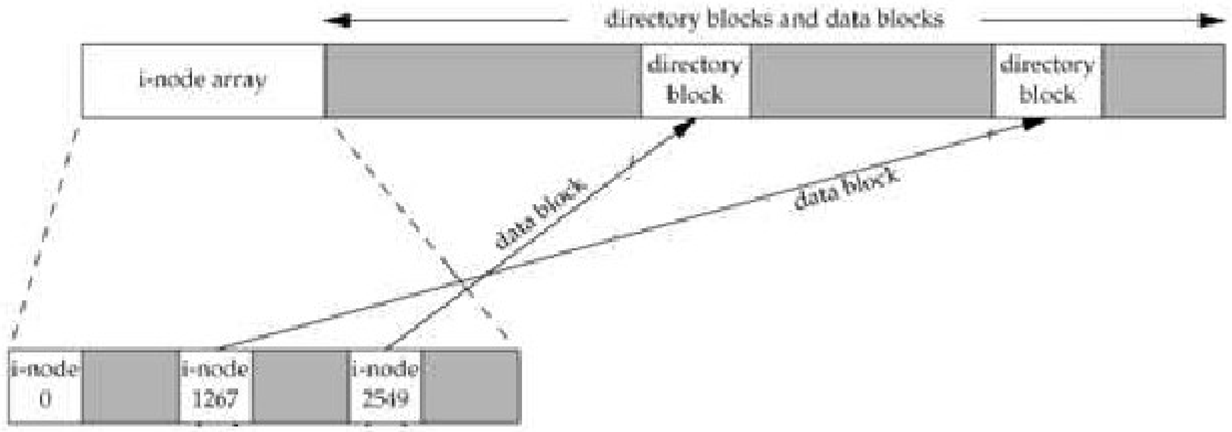
File Systems:Recap

- a directory entry is really just a *hard link* mapping a “filename” to an inode
- you can have many such mappings to the same file



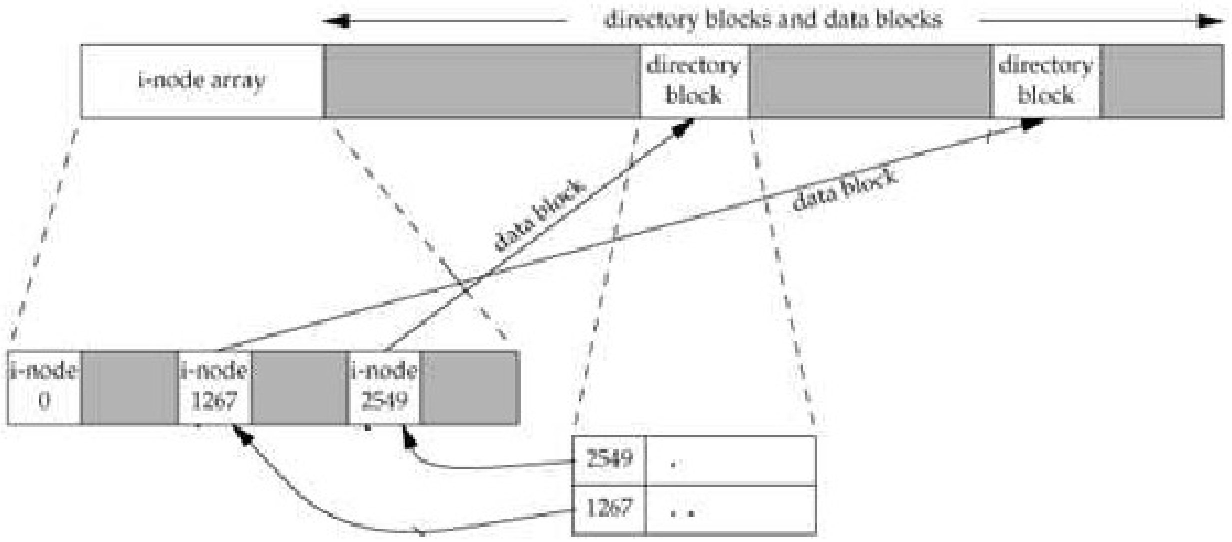
File Systems:Recap

- directories are special "files" containing hardlinks



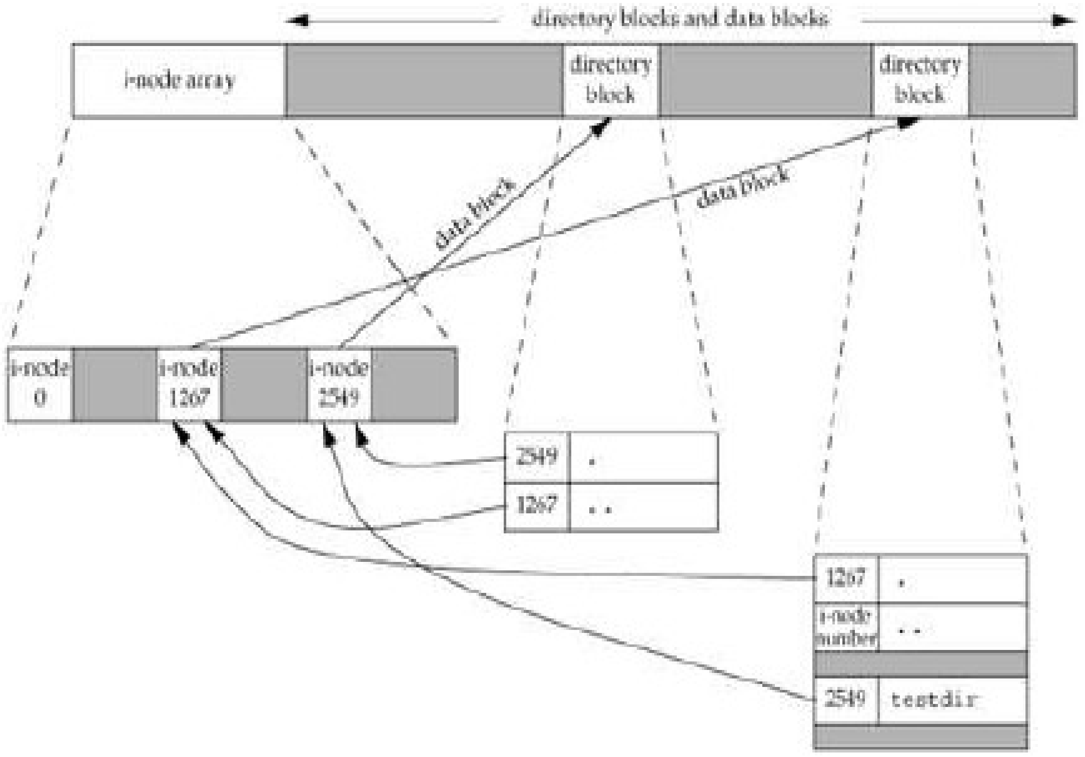
File Systems:Recap

- each directory contains at least two entries:
  - . (*this* directory)
  - . . (the parent directory)



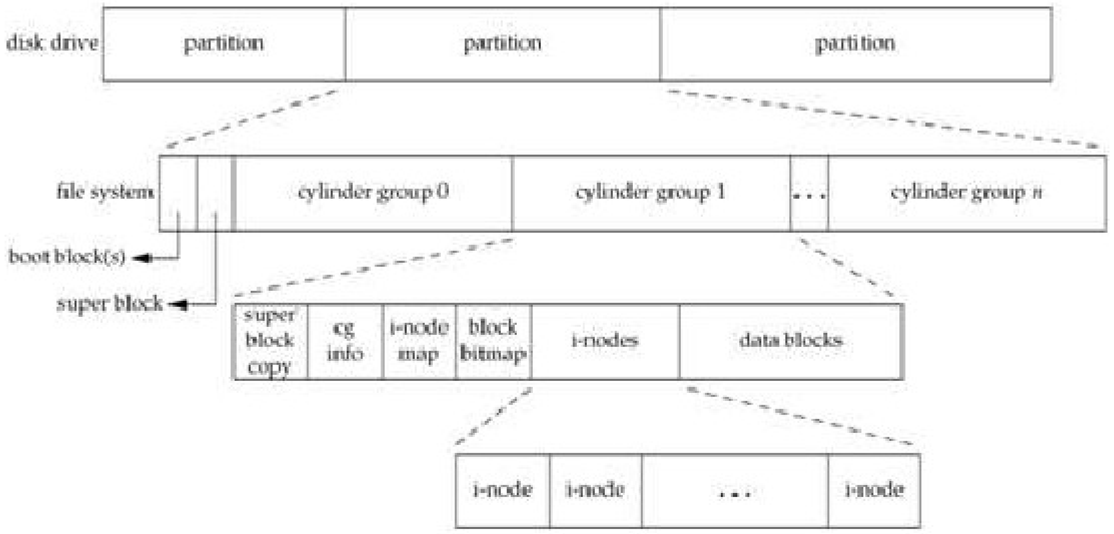
File Systems

- the link count (`st_nlink`) of a directory is at least 2

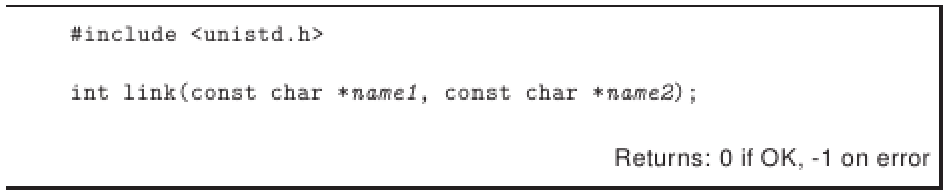


File Systems

- the *inode* contains most of information found in the `stat` structure.
- every *inode* has a *link count* (`st_nlink`): it shows how many "things" point to this inode. Only if this *link count* is 0 (and no process has the file open) are the *data blocks* freed.
- *inode* number in a directory entry must point to an *inode* on the same file system (no hardlinks across filesystems)



link(2)

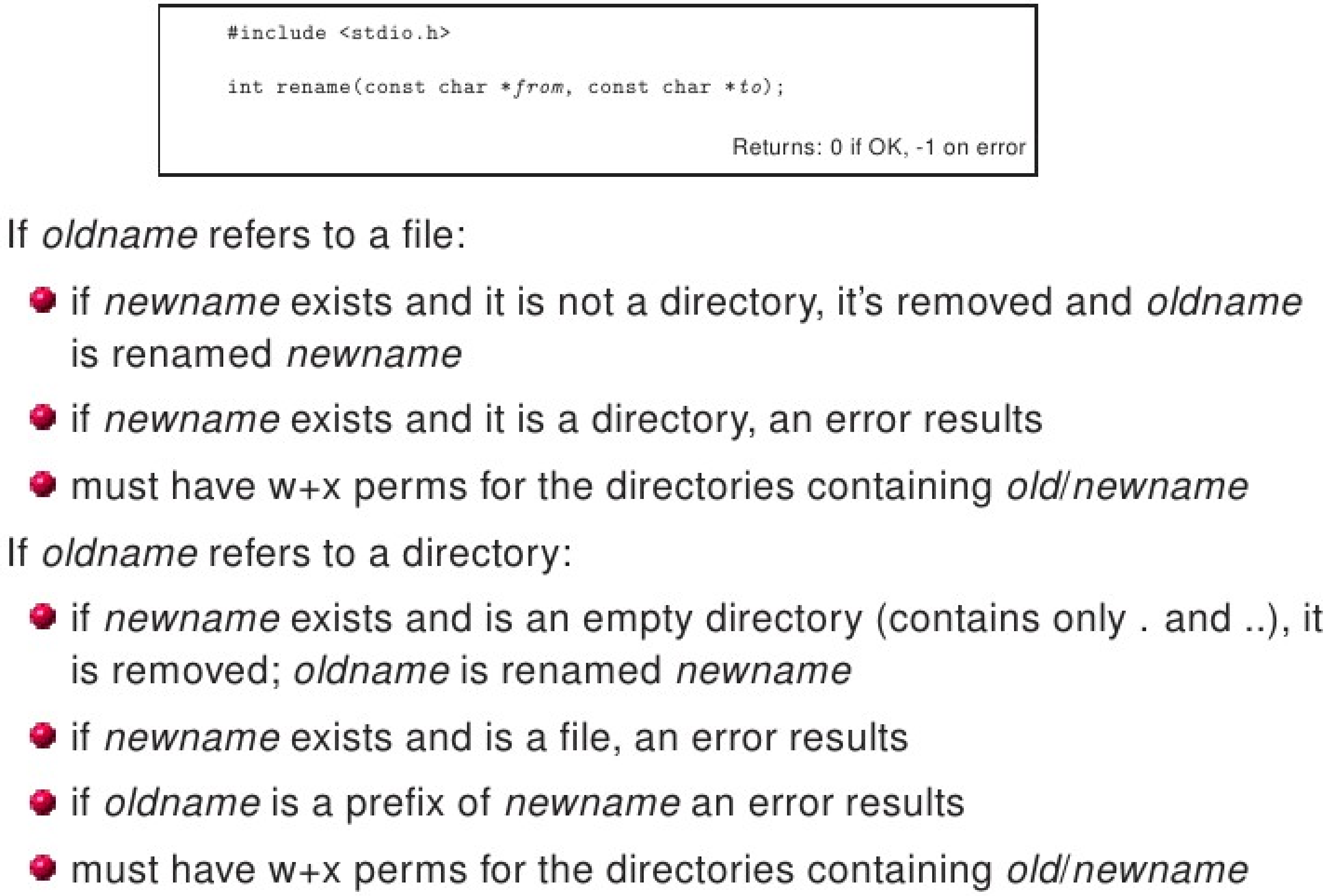


```
#include <unistd.h>

int link(const char *name1, const char *name2);
```

Returns: 0 if OK, -1 on error

- Creates a link to an existing file (hard link).
- waitulink

rename(2)



```
#include <stdio.h>

int rename(const char *from, const char *to);
```

Returns: 0 if OK, -1 on error

- If *oldname* refers to a file:
  - if *newname* exists and it is not a directory, it's removed and *oldname* is renamed *newname*
  - if *newname* exists and it is a directory, an error results
  - must have w+x perms for the directories containing *old*/*newname*

  If *oldname* refers to a directory:
  - if *newname* exists and is an empty directory (contains only . and ..), it is removed; *oldname* is renamed *newname*
  - if *newname* exists and is a file, an error results
  - if *oldname* is a prefix of *newname* an error results
  - must have w+x perms for the directories containing *old*/*newname*