# The Unix File System

## A file is a file is a file...

- Almost everything is represented as files in Unix: regular files, directories, output device (monitor,printer), input device (keyboard), pipes, network sockets...
- This consistency is what makes Unix programs powerful.
- For instance, the cat command that usually displays the contents of a file on your screen can be made to print the file on the printer:

```
Display on screen:
$cat hello.cpp
#include <stdio.h>

int main(void)
{
      printf("hello world!\n");
}
$

Print it:
$lpr < cat hello.cpp
```

cat doesn't know the difference between the monitor and printer. It's just writing its output to the standard place: the standard ouput (stdout). In the first case, the stdout points to the screen, and in the second case, it has been *REDIRECTED* to the standard input (stdin) of the printer.

## Input redirection

- UNIX commands typically input from stdin and output to stdout
- Manipulate stdin and stdout to accomplish other tasks
- Use < to redirect input: lpr < cat hello.cpp
- Use > to redirect output: cat hello.cpp > file2
- Use | (pipe) to connect stdout to stdin: cat hello.cpp | wc -l
- >> appends to existing files: echo more text >> file2

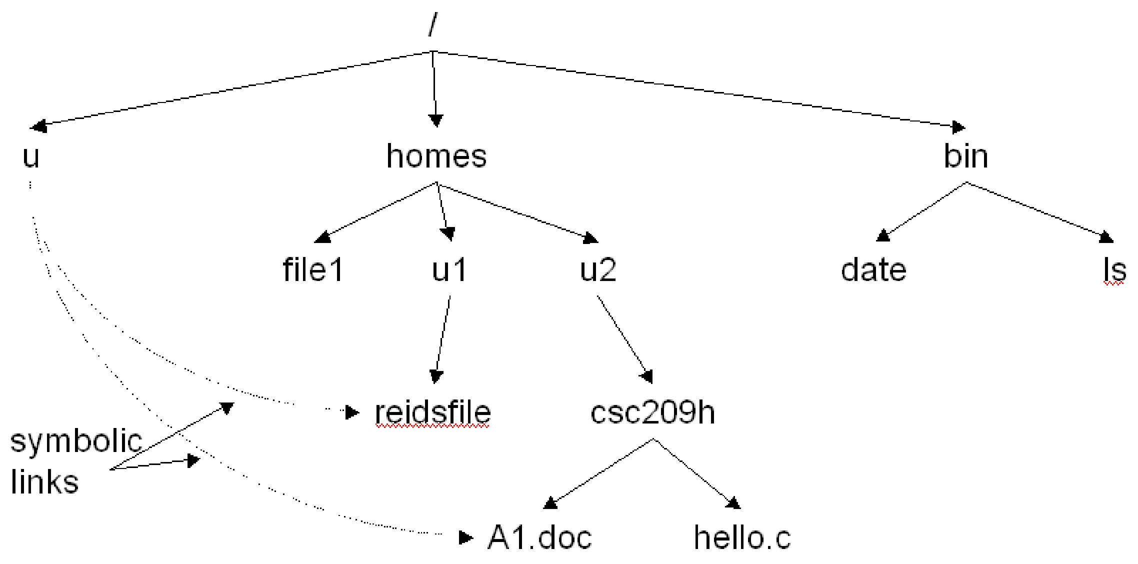
```
$cat hello.cpp > file2
$cat file2
#include <stdio.h>

int main(void)
{
      printf("hello world!\n");
}
$cat file2 | wc -l
     6
$echo more text >> file2
$cat file2
#include <stdio.h>

int main(void)
{
      printf("hello world!\n");
}
more text
$
```

## The Unix File "Tree"

- / represents the root directory
- Each directory contains 0 or more files and subdirectories
- Fully qualified path names begin with root (/) followed by 0 or more subdirectories
- Cannot use / character in filename
- Also don't use '-' (options), '*' and '?' (wild cards) in filenames

The '-' character is often used to pass parameters to commands. For example ls -l gives the detailed listing of files in the current directory. If you had a file that began with '-', ls and other commands will get very confused. For a similar reason, '*' and '?' are used as wildcard patterns to match one or more files: ls * lists all files in the directory.

**Example File Tree: (fig 1-1)**



- One file system per disk partition.
- A file system can be ***mounted*** at any point in the directory tree of another file system.
- u might reside on another disk

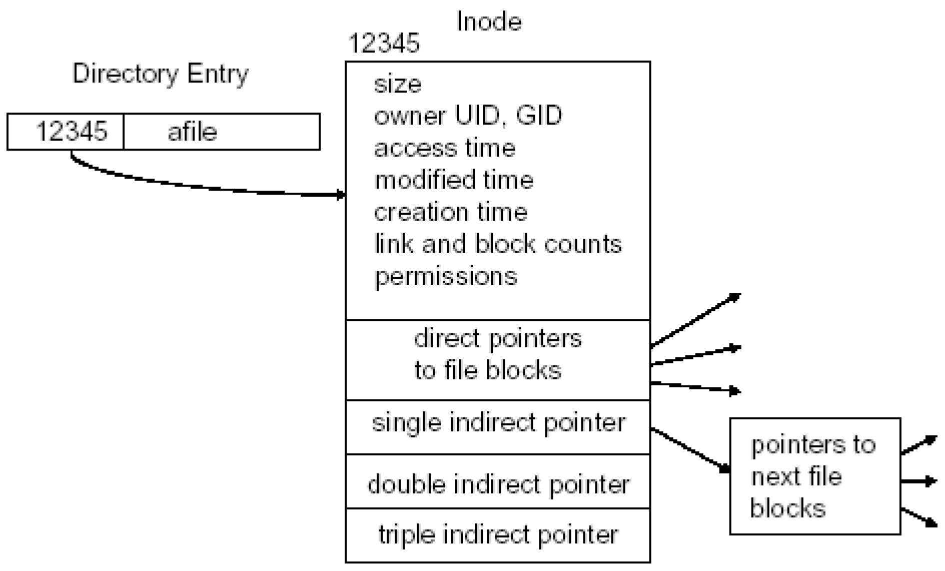
## Fully qualified and Relative path names

- suppose CWD (current working directory) is /usr/you/
- when you type cat hello.c, hello.c is the *relative path*
- use relative path and CWD to construct fully qualified path:
  /usr/you/hello.c

## Under the covers: Inodes:

- A directory is really just a special kind of file that contains a list of filenames and *inode* numbers.
- Many to 1 mapping from directory entry to inode (links, explained later)
- The *inode* is the building block of the UNIX file system. They contain all kinds of information about the directory entry, such as:
  - The access permission
  - The owner and group of the file
  - The last access time, modification time
  - A list of the disk block numbers where data for the file is stored. If the file is small, direct block numbers are stored. Larger files require the inode to store block numbers for blocks that contain more block numbers (indirection).

(Fig 1-2)



An example "i-list" for the directory tree in fig1-1:

| inode # | contents |
|---|---|
| 2 | bin 3, homes 6 |
| 3 | date 4, ls 5 |
| 4 | *contents of date program* |
| 5 | *contents of ls program* |
| 6 | file1 7, u1 8, u2 9 |
| 7 | *contents of file1* |
| 8 | reidsfile 10 |
| 9 | csc209h 11 |
| 10 | *contents of reidsfile* |
| 11 | A1.doc 12, hello.c 13 |
| 12 | *contents of A1.doc* |
| 13 | *contents of hello.c* |
| 14 | |
| 15 | |

the /u/ directory is on another disk, so no inode entry.

## Hard Links

- Hard link links directory entry to the inode describing the actual item
- Each file has 1 or more hard links.
- Removing a file decrements the link count (at 0, the inode & disk space are freed)
- hard links have equal status. Two links to a file: remove either one, the other one stays.
- create hard links with ln

```
$ ls -li
1097747 -rw-r--r--   1 ken    ken         83 Apr 25 00:18 file1
$ ln file1 file2
$ ls -li
1097747 -rw-r--r--   2 ken    ken         83 Apr 25 00:18 file1
1097747 -rw-r--r--   2 ken    ken         83 Apr 25 00:18 file2
$ cat file1
some text1
$ echo add a line >> file1
$ cat file2
some text1
add a line
```

The i in ls -li tells ls to print the inode # (1st column). The 3rd column is the link count. Notice how it changes from 1 to 2 after the additional link has been created with ln file1 file2. Note also how a change to file1 affects file2 as well.

## Soft Link (Symlink)

- A Symlink is a file whose content is treated as path name
- File it points to need not exist
- Remove symlink ≠ remove file
- Symlinks can cross filesystems... hard links can't!

## Programming the File System

## File interfaces in Unix

- Unix has two main mechanisms for managing file access.
- file pointers: standard I/O library (Ch. 11)
  - You deal with a pointer to a FILE structure that contains a file descriptor and a buffer.
  - Use for regular files (more abstract and portable)
- file descriptors: low-level (Ch. 2)
- Each open file is identified by a small integer.
  - Use for pipes, sockets.

You should use the standar I/O library whenever possible. This will make your programs more portable across different platforms. Use file descriptors only when your program requires Unix specific features. In general, structure your code so you can isolate platform dependent code - porting programs with platform dependent code littered everywhere can be a nightmare.

## stdin,stdout,stderr

- 3 files are automatically opened for any executing program:

| | stdio name | File descriptor |
|---|---|---|
| Standard input | `stdin` | 0 |
| Standard output | `stdout` | 1 |
| Standard error | `stderr` | 2 |

- Reading from stdin by default comes from the keyboard
- Writing to stdout or stderr by default goes to the screen.

## File manipulation primitives

| | |
|---|---|
| open() | opens a file for read/write, or create empty file |
| close() | must close opened files |
| read() | |
| write() | |
| lseek() | move to a specific position in a file |
| unlink() | remove a file |
| fcntl() | controls file attributes |

## open()/close()

- int open(const char * pathname,int flags, [mode_t mode]);
- pathname: any valid pathname, relative or absolute
- flags: defined in fcntl.h

| **Flag** | **Meaning** |
|---|---|
| O_RDONLY | read only |
| O_WRONLY | write only |
| O_RDWR | read write |
| O_CREAT | create if doesn't exist |
| O_APPEND | append to the end of the file |
| O_TRUNC | truncate size to 0 if file exists |
| O_EXCL | fail if file exists already |

- mode: Sets the permission of created files. Needed it only if creating files. See notes later in this section.
- int close(int filedes); Make sure to close files you opened!

There are more...look at the detailed doc for open()

example:

```
#define PERMISSION          0644 //don't worry about this yet.

const int ERROR_CODE = -1;
const int EXIT_OK = 0;
const int EXIT_ERROR = 1;
```

```
char * myfile="mynewfile";
int filedes;
filedes = open(myfile,O_WRONLY|O_CREAT,PERMISSION);
if(filedes==ERROR_CODE)
{
        printf("couldn't create file!\n");
        exit(EXIT_ERROR);
}
printf("File opened/created.  Proceeding with program...\n");
.
.
.
//be sure to close any file descriptors you open!
close(filedes);
```

## read()/write()

- ssize_t read(int filedescriptor,void * buffer, size_t n);
- ssize_t write(int filedescriptor,const void * buffer,size_t n);
- most efficient when moving data in blocks that are multiple of disk block size (512)
- read/write in big chunks where possible, to minimize system calls (can get expensive)

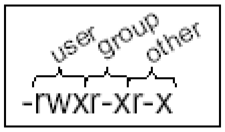
Do not mix in read()/write() calls with ANSI standard IO calls like fread(), fwrite() etc. for the same file. This is because the ANSI standard IO calls provide for buffering, and so the actual read or write from disk may not occur immediately after you make the call. By contrast read()/write() reads or writes immediately, so mixing the two types of I/O routines for the same file is a dangerous practice.

## lseek() - random access

- off_t lseek(int filedes,off_t offset,int start_flag);
- takes the usual file descriptor
- offset is an signed int measured with respect to start_flag, cast it to off_t
- start_flag can be 1 of:
    SEEK_SET: offset is from begining of file
    SEEK_CUR: offset is from current location
    SEEK_END: offset is from end of file

## File/Directory permissions

```
-rwxr-xr-x 1 reid   0 Jan    11 22:26        allexec*
-rw-r--r--          1 reid   0 Jan 11 22:23  allread
-rw-------          1 reid   0 Jan 11 22:23  ownerread
-r--r--r--   1 reid 0 Jan    11 22:23        readonly
drwxr-xr-x1 reid    0 Jan    11 22:23        dir-read
drwx--x--x          1 reid   0 Jan 11 22:23  dir-search
```



## Understanding File Permissions (From the notes of Karen Reid)

Although the same letters are used to denote permissions for regular files and for directories, they have slightly different meanings.

| **Attribute** | **File** | **Directory** |
|---|---|---|
| r | read | read |
| w | write | create/remove |
| x | execute | search |

For regular files, r means that the file can be displayed by cat, w means that it can be changed by an editor, and x means that is can be executed as a program.
For example:

```
-rwxr-xr-x   1 reid           0 Jan 11 22:26 allexec*
-rw-r--r--   1 reid           0 Jan 11 22:23 allread
-rw-------   1 reid           0 Jan 11 22:23 ownerread
-r--r--r--   1 reid           0 Jan 11 22:23 readonly
```

- Everyone can read `allread` and `readonly`.
- Everyone can execute `allexec`.
- Only the owner of `ownerread` can read and write it.
- If the owner of `readonly` tries to edit it the editor will not allow the owner to save the changes.
- If anyone tries to execute `allread` (or `ownerread`, or `readonly`) they will get the message "Permission denied."

A directory is a file that stores a list of filenames along with the inode number of the file. (The inode is a data structure that stores data about a file and pointers to disk blocks that hold the file's data.)
Read permission for a directory means that **ls** can read the list of filenames stored in that directory. If you remove read permission from a directory, **ls** won't work.

For example:

```
        % ls -ld testdir
        d-wx--x--x   2 reid          512 Jan 11 22:41 testdir/
        % ls -l testdir
        testdir unreadable
        % cat testdir/hidefile
        I can still read this file
```

Note that it is still possible to read and write files inside testdir even though we cannot list its contents.
Write permission for a directory means that you can create and remove files from that directory. You should never allow write permission for others on any of your directories. If you did, any other user could remove all of your files from such a directory.

```
        % ls -l
        -rw-r--r--   1 reid            0 Jan 11 22:23 allread
        dr-xr-xr-x   2 reid          512 Jan 11 22:42 testdir/
        % cp allread testdir
        cp: cannot create testdir/allread: Permission denied
```

When write permission is removed from `testdir`, I can no longer copy a file into `testdir`.
Execute permission for a directory means that you can "pass through" the directory in searching for subdirectories. You need to have executable permissions for every directory on a path to run a command on that path or to used that path as an argument for another command.
For example, when I remove execute permission from `testdir` I can no longer really do anything with files and directories under `testdir`

```
        % ls -l testdir/
        total 2
```

```
-rw-r--r--   1 reid           28 Jan 11 22:42 hidefile
drwxr-xr-x   2 reid          512 Jan 11 23:02 subdir/
% ls -l testdir/subdir
total 1
-rwxr--r--   1 reid          136 Jan 11 23:02 var.sh*
% chmod a-x testdir
% ls -ld testdir
drw-r--r--   3 reid          512 Jan 11 23:02 testdir/
% testdir/subdir/var.sh
testdir/subdir/var.sh: Permission denied.
% cd testdir
testdir: Permission denied.
% ls testdir/subdir
ls: testdir/subdir: Permission denied
% cat testdir/hidefile
cat: cannot open testdir/hidefile
```

## Changing permissions

- There are 9 permissions to be set (rwxrwxrwx)
- Idea: use 9 bits, each to represent 1 of the permissions
- These correspond to octals #s:

| Octal value | Binary value | Symbolic constant | Meaning |
|---|---|---|---|
| 0400 | **100000000** | S_IRUSR | Read allowed by owner |
| 0200 | **010000000** | S_IWUSR | Write allowed by owner |
| 0100 | **001000000** | S_IXUSR | Execute allowed by owner |
| 0040 | **000100000** | S_IRGRP | Read allowed by group |
| 0020 | **000010000** | S_IWGRP | Write allowed by group |
| 0010 | **000001000** | S_IXGRP | Execute allowed by group |
| 0004 | **000000100** | S_IROTH | Read allowed by all other users |
| 0002 | **000000010** | S_IWOTH | Write allowed all other users |
| 0001 | **000000001** | S_IXOTH | Execute allowed by all other users |

## Combining permissions

- Use bitwise OR to combine the bits(flags)
- or add the octal values:
  Execute has value 1
  write has vale 2
  read has value 4
- add for each class of user, the values to be granted (The left most digit is for user, the middle for group, and the right most digit is for other)

eg. rwxrwxrwx = 0777
rw-rw-r-- = 0664
rwx--x--x = 0711

These octal values will come in handy when programming the file system, as we will shortly see. However, they are also useful on the command line. You can use the chmod command to change the permission on a file using the octal values:

```
chmod 711 myfile
```

will set myfile's permission to rwx--x--x. If you don't want to calculate the octal values, there's an easier way to change the permission using chmod: give chmod one or more of the characters 'u','g','o' (user, group, others) followed by one of '+', '-', and '=' and finally one or more of 'r','w', and 'x'. So to add write permission to myfile for all users:

```
chmod ugo+w myfile
```

To turn on all permissions for all users:

```
chmod ugo+rwx myfile
```

## Open() revisited

- recall int open(const char * pathname,int flags, [mode_t mode]);
- if open() needs to create a file (because O_CREAT was specified), the 3rd parameter mode is required.
- Pass it the octal value we talked about above or combine the flags (S_IRUSR,S_IWUSR etc.) with bitwise OR.

  example: open(myfile,O_WRONLY|O_CREAT,0755); If creating file, set file permission to rwxr-xr-x.

- Must have 0 infront of 755, otherwise compiler thinks it's decimal!

## File creation mask

- if a permission bit in the file creation mask is set, then that permission is always turned OFF when a file is created.
- so open(myfile,0_WRONLY|O_CREAT,mode); is actually the same as open(myfile,O_WRONLYO_CREAT,(~mask)&mode);
- set the file creation mask with umask()
- set file creation mask to 0 if want permissions to be exactly what you ask for.

## stat()

- Returns much of the information about a file that's stored in its inode entry:
  - inode number
  - access permissions
  - #of links
  - owner of file, group of file
  - acess times
  - ...and others

See detailed documentation

## Other useful file functions:

| Function Name | What it does |
|---|---|
| link() | create a hard link to a file |
| unlink() | decrements the link count on a file (when it reaches 0, the file is removed) |
| symlink() | creates a symbolic link to a file |
| readlink() | read the contents of a symbolic link |
| access() | check for file access permissions and existence |
| chmod() | change the permissions on a file |
| chown() | change the owner of a file |
| rename() | renames a file |

## Useful Directory functions:

| Function Name | What it does |
|---|---|
| mkdir() | creates a new directory |
| rmdir() | removes an existing directory.  Directory Must be empty |
| opendir() | returns a DIR * (pointer to DIR structure) used other directory manipulation calls |
| closedir() | close the DIR * returned by opendir() |
| readdir() | returns one entry of a directory at a time |
| rewinddir() | causes readdir() to restart from the beginning of the directory |
| chdir() | change the current directory |
| getcwd() | get the current working directory |
| ftw() | "file tree walk" – continually calls a function you supply until all items in a directory and all its sub-directories have been exhausted. |
| pathconf()<br>fpathconf() | returns useful system limits such as maximum length of filename and pathname |

An example:

```
char *name, line[LINESIZE], *lp; int len;
DIR *dp;  struct dirent *entry;   FILE *fp;
name = argv[1];
len = strlen(name);
dp = opendir(".");
for (entry = readdir(dp); entry != NULL;entry = readdir(dp))
    if ((strncmp(name, entry->d_name, len)) == 0) {
          fp = fopen(entry->d_name, "r");
          lp = fgets(line, LINESIZE, fp);
          fprintf(stdout, "%s: %s", entry->d_name, lp);
    }
closedir(dp);
```