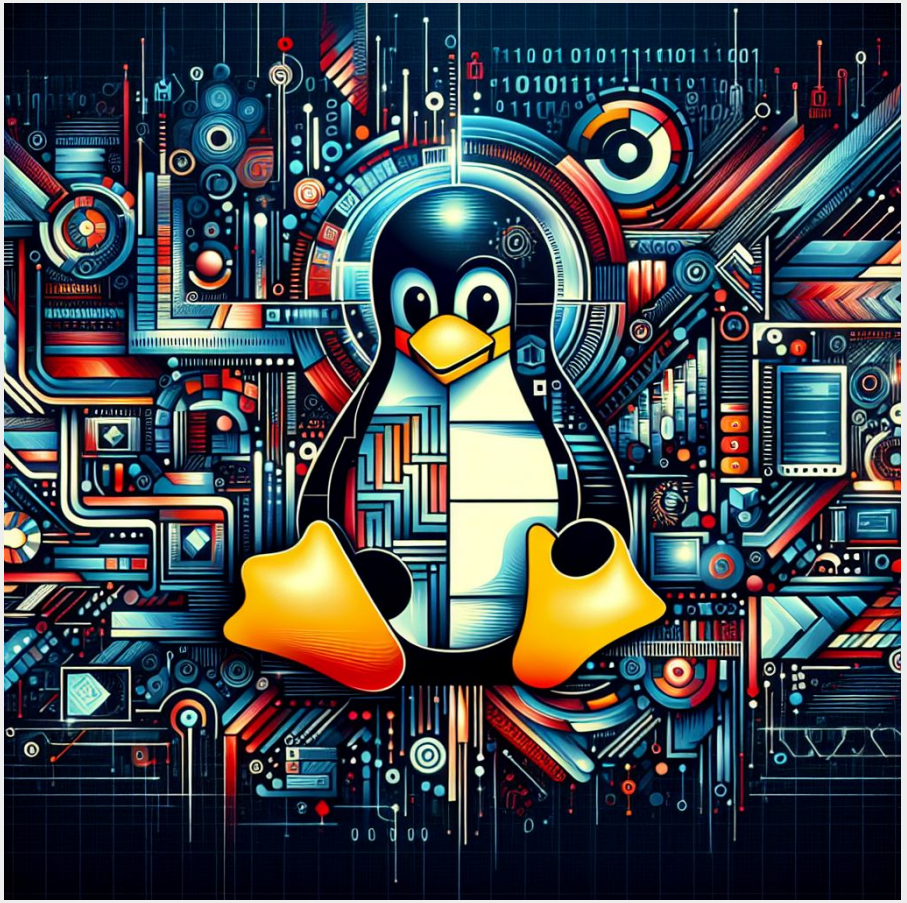
UNSW SYDNEY

COMP1521 24T2 Lec 11/12

File Systems Part 2



2024

Angela Finlayson

Format from Hammond Pearce, material from COMP1521

IO Performance & Buffering libc vs stdio

Let's compare our implementations of cp!

```
$ clang -O3 cp_x.c -o cp_x
$ dd bs=1M count=10 </dev/urandom >random_file
10485760 bytes (10 MB, 10 MiB) copied, 0.183075 s, 57.3 MB/s
$ time ./cp_x random_file random_file_copy
```

Can we get any insights from strace?

```
$strace ./cp_x random_file random_file_copy
```

Compare:

Linux cp command, cp_fgetc_one_byte.c, cp_libc_one_byte.c, cp_libc.c

stdio.h buffering

stdio functions use buffering to improve efficiency

Goal: reduce number of system calls (expensive)

Does a **read** to fill whole buffer

- can read next byte from buffer
- does not do another read till it needs data not in buffer

Delays calls to **write**

- stores data in buffer (array) instead
- calls write when buffer is full, file is closed,
- newline is encountered (for line buffered output - e.g. output to screen).
- `int fflush(FILE *stream); //manually flush the buffer`
  `//i.e. force the write`

Seeking with libc system call wrapper

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

- change the **current position** in given stream
- **offset** is in bytes, and can be negative
- **whence** can be one of
  - SEEK_SET : set **offset** from start of file
  - SEEK_CUR: set file **offset** from current position
  - SEEK_END: set file **offset** from end of file

- seeking beyond end of file leaves a gap which reads as 0's
- seeking back beyond start of file sets position to start of file
- example: `lseek(fd, -1, SEEK_END); // move to before last byte in file`

Seeking with stdio.h

```
int fseek(FILE *stream, long offset, int whence);
```

- is stdio equivalent to **`lseek()`** except:
  - requires a FILE * input instead of int file descriptor
  - influences stdio buffers
  - returns 0 or -1 for error

```
fseek(stream, 42, SEEK_SET); // move to after 42nd byte
fseek(stream, 58, SEEK_CUR); // 58 bytes forward from current position
fseek(stream, -7, SEEK_CUR);  // 7 bytes backward from current position
fseek(stream, -1, SEEK_END);  // move to before last byte in file
```

```
long ftell(FILE *stream);  //return current file position
```

Demo code fseek.c and fuzz.c and advanced example: create_gigantic_file.c

File Systems

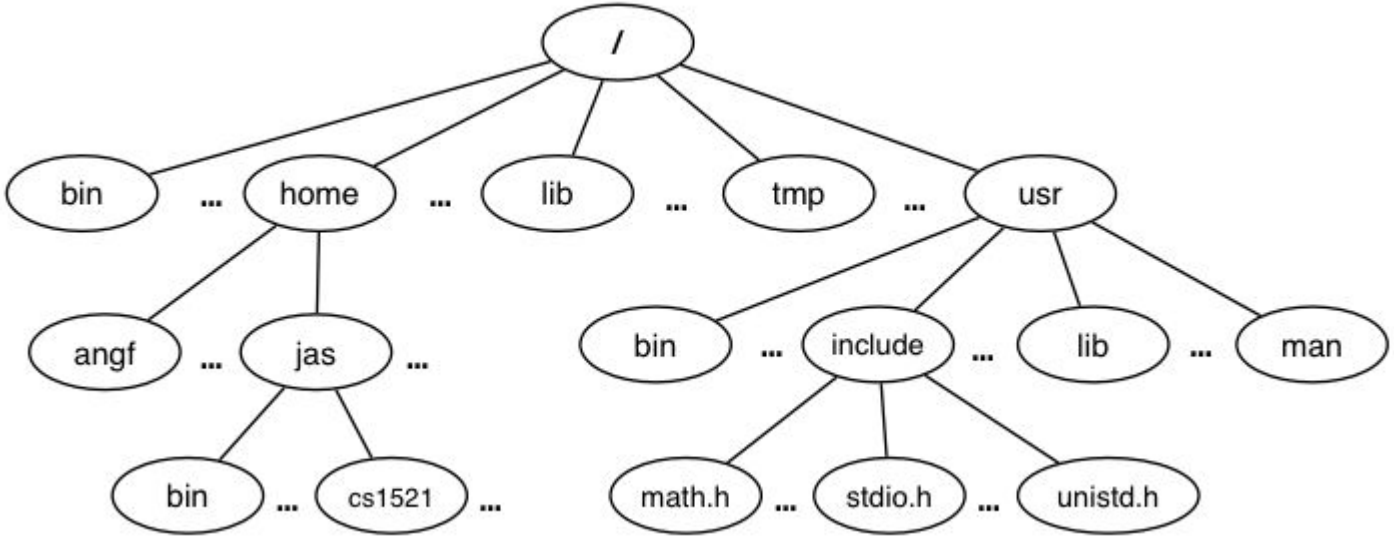
File systems manage stored data (e.g. on disk, SSD)

File = named sequence of bytes, stored on device

- file system maps name to location on device
- file system maintains meta-data (e.g. permissions, time stamps)

Directory = sets of files or directories

Unix/Linux File System



Unix/Linux file system is tree-like

- symlinks actually make it into a graph
- if traversing you may infinitely loop if following them

Unix-like File Names

Sequences of 1 or more bytes

- filenames can contain any byte except
- **0x00** bytes (ASCII '\0') used to terminate filenames
- **0x2F** bytes (ASCII '/') used to separate components of pathnames.
- maximum filename length, depends on file system, typically 255

Two filenames have a special meaning:

- **.** current directory
- **..** parent directory

Some programs (shell, ls) treat filenames starting with **.** specially.

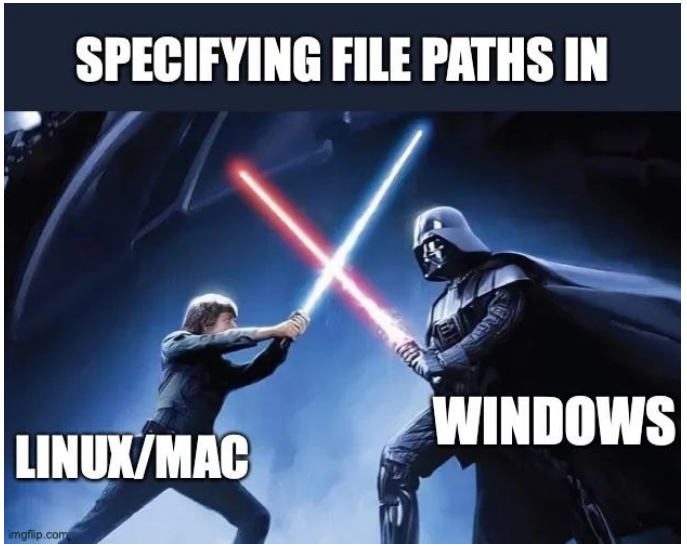
Paths and directories

Absolute pathnames start with a leading **/** and give full path from root e.g.

- **/usr/include/stdio.h**

Relative pathnames do **not** start with a leading **/** e.g.

- **../../another/path/prog.c**
- **./a.out**
- **main.c**



Current Working Directory

Every process (running program) has a **current working directory (CWD)**

- this is an absolute pathname
- this is the directory from where the process was run from
- shell command **pwd** prints the **CWD**

Relative pathnames appended to CWD of process e.g.

- if CWD is **/home/z5555555/lab07/**
- and relative path is **main.c**
- absolute path would be **/home/z5555555/lab07/main.c**

Unix-like File Metadata

Metadata for file system objects is stored in **inodes**, which hold

- location of file contents in file systems
- file type (regular file, directory, ...)
- file size in bytes
- file ownership
- file access permissions - who can read, write, execute the file
- timestamps - times of file was created, last accessed, last updated

Inodes

Files system has large table of inodes containing metadata

- Inode-number is the inodes id
    - Unique for file system like zid within UNSW

Directories are effectively a collection of (name, inode-number) pairs

- `ls -i` prints **inode-numbers**

File Access: Behind the scenes

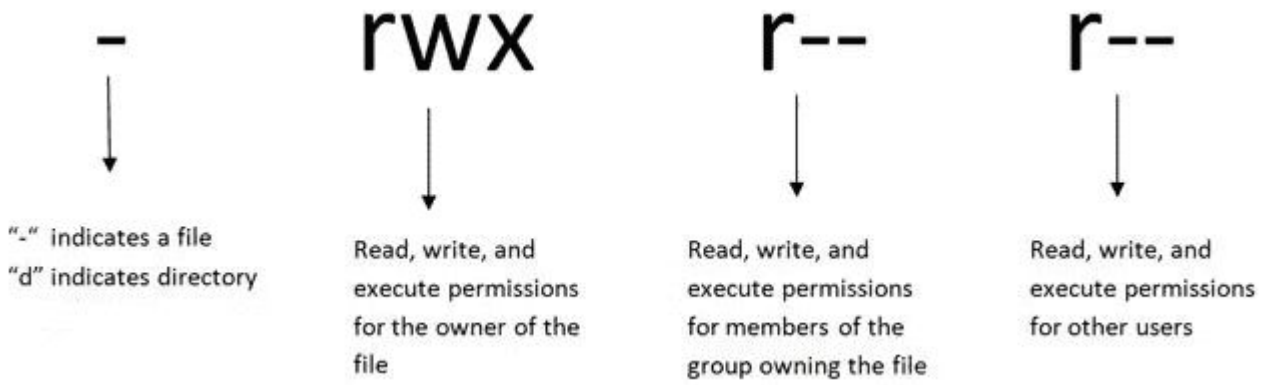
Access to files by name proceeds (roughly) as...

- **open directory** and scan for **name**
    - if not found, "No such file or directory"
- if found as (**name,inumber**), **access inode table** inodes[**inumber**]
- collect **file metadata** and...
    - check file access permissions given current user/group
        - if don't have required access, "Permission denied"
    - update access timestamp
- use data in inode (size location) to **access file** contents

File Permissions

Every file and directory in linux has read, write and execute permissions (access rights) for each of the following user groups:

- user: the file's owner
- group: the members of the file's group
- other: everyone else
- type **`ls -l`** on command line to see



File Permissions: read, write, execute

	Read	**Write**	**Execute**
File	View contents of file	Modify file	Run as executable
Directory	View names of file e.g. use ls	Create, delete, rename files	Can cd into it. Also needed to access (read, write, execute) items in directory

Modifying Permissions

You can think of permissions as a set of bits and then each 3 bits as an octal digit. e.g.

```
  rwx r-x r-x
  111 101 101
    7   5   5
```

You can use the **chmod** command to set the permissions of a file or directory using the desired 3 digit octal code. e.g.

```
$ chmod 700 f.txt
```

Hard Links and Symbolic Links

File system **links** allow multiple paths to access the same file

- Hard links
    - multiple names referencing the same file (inode)
    - the two entries must be on the same filesystem
    - can not create a (extra) hard link to directories
    - all hard links to a file have equal status
    - file destroyed when last hard link removed
    - e.g. Assuming 'fileA' already exists:

        ```
        ln fileA fileB
        ```

        would create a hard link named 'fileB'

Hard Links and Symbolic Links

File system **links** allow multiple paths to access the same file

- Symbolic links (symlinks)
    - point to another path name
    - accessing the symlink (by default) accesses the file being pointed to
    - symbolic link can point to a directory
    - symbolic link can point to a pathname on another filesystems
    - symbolic links don't have permissions (not needed - they are just a pointer)
    - e.g. Assuming 'fileA' already exists:

```
ln -s fileA fileB
```

would create a symbolic link named 'fileB'

C library wrapper for stat system call

```
int stat(const char *pathname, struct stat *statbuf);
```

- returns metadata associated with **pathname** in **statbuf**
- metadata returned includes:
    - inode number
    - type (file, directory, symbolic link, device)
    - size of file in bytes (if it is a file)
    - permissions (read, write, execute)
    - times of last access/modification/status-change
- returns **-1** and sets **errno** if metadata not accessible

C library wrapper for stat system call

```
int lstat(const char *pathname, struct stat *statbuf);
```

- same as stat() but doesn't follow symbolic links
    - in other words gives you metadata about the symbolic link and not the file it links to
    - important not to get stuck in infinite loops

```
int fstat(int fd, struct stat *statbuf);
```

- same as stat() but gets data via an open file descriptor

See **man 2 stat**
man 3 stat
man 7 inode

definition of struct stat

man 3 stat

```
struct stat {
  dev_t     st_dev;      /* ID of device containing file */
  ino_t     st_ino;      /* Inode number */
  mode_t    st_mode;     /* File type and mode */
  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 */
  ...
};
```

st_mode field of struct stat

man 7 inode

st_mode is a bitwise-or of these values (& others):

```
S_IFLNK    0120000   symbolic link
S_IFREG    0100000   regular file
S_IFDIR    0040000   directory
S_IRUSR    0000400   owner has read permission
S_IWUSR    0000200   owner has write permission
S_IXUSR    0000100   owner has execute permission
S_IRGRP    0000040   group has read permission
S_IWGRP    0000020   group has write permission
S_IXGRP    0000010   group has execute permission
S_IROTH    0000004   others have read permission
S_IWOTH    0000002   others have write permission
S_IXOTH    0000001   others have execute permission
```

Code demos stat.c

stat0.c

stat.c

Good sample program at bottom of man 2 stat

Making a directory

```
int mkdir(const char *pathname, mode_t mode);
```

returns 0 if successful, returns -1 and sets `errno` otherwise

- for example: `mkdir("newDir", 0755)`

if **pathname** is e.g. `a/b/c/d`

- all of the directories `a`, `b` and `c` must exist
- directory `c` must be writable to the caller
- directory `d` must not already exist

the new directory contains two initial entries

- `.` is a reference to itself
- `..` is a reference to its parent directory

Demo: mkdir.c

Opening and Reading directories

// open a directory stream for directory name

```
DIR *opendir(const char *name);
```

// return a pointer to next directory entry

```
struct dirent *readdir(DIR *dirp);
```

// close a directory stream

```
int closedir(DIR *dirp);
```

Found in man 3

Demo list_directory.c

Useful Linux (POSIX) functions

`chmod(char *pathname, mode_t mode)` // change permission of file/...

`unlink(char *pathname)` // remove a file...

`rename(char *oldpath, char *newpath)` // rename a file/directory

`chdir(char *path)` // change current working directory

`getcwd(char *buf, size_t size)` // get current working directory

`link(char *oldpath, char *newpath)` // create hard link to a file

`symlink(char *target, char *linkpath)` // create a symbolic link

Demo: chmod.c rm.c rename.c my_cd.c getcwd.c nest_directories.c many_links.c chain_links.c

Everything is a File

Originally files only managed data stored on a magnetic disk.

Unix philosophy is: **Everything is a File**

File system used to access:

- files
- directories (folders)
- storage devices (disks, SSD, ...)
- peripherals (keyboard, mouse, USB, ...)
- system information
- inter-process communication
- network