COMP1521 25T2 — Files

https://www.cse.unsw.edu.au/~cs1521/25T2/

Operating system - What Does it Do.

- Operating system sits between the user and the hardware.
- Operating system effectively provides a virtual machine to each user.
- This virtual machine is much simpler than a real machine
  - much easier for user to write code
  - difficult (bug-prone) code implemented by operating system
- The virtual machine interface can stay the same across different hardware.
  - much easier for user to write portable code which works on different hardware
- Operating systems can coordinate/share access to resources between users.
- Operating systems can provide privileges/security.

Operating System - What Does it Need from Hardware.

- needs hardware to provide a **privileged** mode
  - code running in privileged mode can access all hardware and memory
  - code running in privileged mode has unlimited access to memory
- needs hardware to provide a **non-privileged** mode which:
  - code running in non-privileged mode can not access hardware directly
  - code running in non-privileged mode has limited access to memory
  - provides mechanism to make requests to operating system
- operating system (kernel) code runs in **privileged** mode
- operating system runs user code in **non-privileged** mode
  - with memory access restrictions so user code can only use memory allocated to it
- user code can make requests to operating system called **system calls**
  - a system call transfers execution to operating system code in privileged mode
  - at completion of request operating system (usually) returns execution back to user code in non-privileged mode

System Call - What is It

- system call allow programs to request hardware operations
- system call transfers execution to OS code in **privileged** mode
  - includes arguments specifying details of request being made
  - OS checks operation is valid & permitted
  - OS carries out operation
  - transfers execution back to user code in **non-privileged** mode
- different operating system have different system calls
  - e.g Linux system calls very different Windows system calls
- Linux provides 400+ system calls
- examples of operations that might be provided by system call:
  - read or write bytes to a file
  - request more memory
  - create a process (run a program)
  - terminate a process
  - send information via a network

System Calls in mipsy

- `mipsy` provides a virtual machine which can execute MIPS programs
- `mipsy` also provides a tiny operating system
- small number of **`mipsy`** system calls for I/O and memory allocation
- access is via the **`syscall`** instruction
  - MIPS programs running on real hardware also use **`syscall`**
  - on Linux **`syscall`**, passes execution to operating system code
  - Linux operating system code carries out request specified in `$v0` and `$a0`
- `mipsy` system calls are designed for students writing tiny MIPS programs without library functions
  - e.g system call **1** - print an integer, system call **5** read an integer
- system calls on real operating systems are more general
  - instead system call might be read ***n*** bytes, write ***n*** bytes
  - users don't normally access system calls directly
  - users call library functions e.g. **`printf`** & **`fgets`** which make system calls, usually via other functions

Experimenting with Linux System Calls

- like `mipsy` every Linux system call has a number, e.g system call **2 is** write bytes to a file
- Linux provides 400+ system calls

```
$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h
...
#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
#define __NR_stat 4
...
#define __NR_pidfd_getfd 438
#define __NR_faccessat2 439
#define __NR_process_madvise 440
```

Aside: The Linux Manual

man

The linux manual (man) is divided into sections.

Important sections for this course include:

1. Executable programs or shell commands eg. ls, cp
2. System calls (we will be looking at many of these in the coming weeks)
3. Library calls eg. strcpy, scanf

And other sections that you can find out about by using the command man man which shows the manual page about the manual.

You can get more information about individual sections by using man 1 intro, man 2 intro etc.

Advice: man will be available in the exam. Get used to using it!

System Calls to Manipulate files

Some important Unix system calls:

- `0` — **`read`** — read some bytes from a file **descriptor**
- `1` — **`write`**— write some bytes to a file **descriptor**
- `2` — **`open`** — open a file system object, returning a **file descriptor**
- `3` — **`close`** — stop using a **file descriptor**
- `4` — **`stat`** — get file system metadata for a pathname
- `8` — **`lseek`** — move **file descriptor** to a specified offset within a file
- above system calls manipulate files as a *stream of bytes* accessed via a ***file descriptor***
  - **file descriptors** are small integers
  - really index to a per-process array maintained by operating system
- On Unix-like systems: a ***file*** is sequence (array) of zero or more bytes.
  - no meaning for bytes associated with file
    - file metadata doesn't record that it is e.g. ASCII, MP4, JPG, …
    - Unix-like files are just bytes

Using a system call to print a message to stdout

- the C function **`syscall`** allows to make a Linux system call without writing assembler
  - **`syscall`** itself is written partly/entirely in assembler
    - e.g.: https://code.woboq.org/userspace/glibc/sysdeps/unix/sysv/linux/x86_64/syscall.S.html
- **`syscall`** is not normally used by programmers in regular C code
  - most system calls have their own C wrapper function, these wrapper function are safer & more convenient
  - e.g. the write system call has a wrapper C function called **`write`**
- we only use **`syscall`** to experiment & learn

```
char bytes[13] = "Hello, Zac!\n";
// argument 1 to syscall is the system call number, 1 is write
// remaining arguments are specific to each system call
// write system call takes 3 arguments:
//   1) file descriptor, 1 == stdout
//   2) memory address of first byte to write
//   3) number of bytes to write
syscall(1, 1, bytes, 12); // prints Hello, Zac! on stdout
```

source code for hello_syscalls.c

Using system calls to copy a file #1 - opening files

```
// cp <file1> <file2> with syscalls and no error handling
int main(int argc, char *argv[]) {
    // system call number 2 is open, takes 3 arguments:
    //   1) address of zero-terminated string containing file pathname
    //   2) bitmap indicating whether to write, read, ... file
    //      O_WRONLY | O_CREAT == 0x41 == write to file, creating if necessary
    //   3) permissions if file will be newly created
    //      0644 == readable to everyone, writeable by owner
    long read_file_descriptor  = syscall(2, argv[1], O_RDONLY,                        0);
    long write_file_descriptor = syscall(2, argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

source code for cp_syscalls.c

Using system calls to copy a file #2 - copying the bytes

```
while (1) {
    // system call number 0 is read -  takes 3 arguments:
    //   1) file descriptor
    //   2) memory address to put bytes read
    //   3) maximum number of bytes read
    // returns number of bytes actually read
    char bytes[4096];
    long bytes_read = syscall(0, read_file_descriptor, bytes, 4096);
    if (bytes_read <= 0) {
        break;
    }
    // system call number 1 is write - takes 3 arguments:
    //   1) file descriptor
    //   2) memory address to take bytes from
    //   3) number of bytes to written
    // returns number of bytes actually written
    syscall(1, write_file_descriptor, bytes, bytes_read);
}
```

source code for cp_syscalls.c

C Library Wrappers for System Calls

- On Unix-like systems there are C library functions corresponding to each system call,
  - e.g. `open`, `read`, `write`, `close`
  - the **`syscall`** function is not used in normal coding
- These functions are not portable
  - C used on many non-Unix operating systems with different system calls
- POSIX standardizes a few of these functions
  - some non-Unix systems provide implementations of these functions
- but better to use functions from standard C library, available everywhere
  - e.g fopen, fgets, fputc from **`stdio.h`**
  - on Unix-like systems these will call `open`, `read`, `write`
  - on other platforms, will call other low-level functions
- but sometimes we need to use lower level non-portable functions
  - e.g. a database implementations need precise control over I/O operations

Extra Types for File System Operations

Unix-like (POSIX) systems add some extra file-system-related C types in these include files:

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

- **off_t** — offsets within files
  - typically **int64_t** - signed to allow backward references
- **size_t** — number of bytes in some object
  - typically **uint64_t** - unsigned since objects can't have negative size
- **ssize_t** — sizes of read/written bytes
  - typically **uint64_t** - similar to **size_t**, but signed to allow for error values
- **struct stat** — file system object metadata
  - stores information *about* file, not its contents
  - requires other types: **ino_t**, **dev_t**, **time_t**, **uid_t**, ...

C library wrapper for open system call

```
int open(char *pathname, int flags)
```

- open file at **pathname**, according to **flags**
- **flags** is a bit-mask defined in **<fcntl.h>**
  - O_RDONLY — open for reading
  - O_WRONLY — open for writing
  - O_APPEND — append on each write
  - O_RDWR — open object for reading and writing
  - O_CREAT — create file if doesn't exist
  - O_TRUNC — truncate to size 0
- flags can be combined e.g. (O_WRONLY|O_CREAT)
- if successful, return file descriptor (small non-negative int)
- if unsuccessful, return **-1** and set **errno** to value indicating reason

errno - why did that system call fail?

- C library has an interesting way of returning error information
- functions typically return **-1** to indicate error
- and set **errno** to integer value indicating reason for error
- these integer values are #define-d in **errno.h**
- see **man errno** for more infomation
- convenient function **perror()** looks at errno and prints message with reason
- or **strerror()** converts errno integer value to string describing reason for error
- **errno** looks like **int** global variable
  - C library designed before multi-threaded systems in common use
  - errno can not really be a global variable on multi-threaded platform
  - each thread needs a separate **errno**
  - clever workaround: **errno #define**d to function which returns address of variable for this thread

C library wrapper for read system call

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

- read (up to) **count** bytes from **fd** into **buf**
  - **buf** should point to array of at least **count** bytes
  - read does (can) not check **buf** points to enough space
- if successful, number of bytes actually read is returned
- `0` returned, if no more bytes to read
- `-1` returned if error and **`errno`** set to reason
- associated with a file descriptor is a ***current position*** in file
- next call to **read()** will return next bytes from file
- repeated calls to reads will yield entire contents of file
- can also modify this current position with **`lseek()`**

C library wrapper for write system call

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

- attempt to write **count** bytes from *buf* into

  stream identified by file descriptor **`fd`**
- if successful, number of bytes actually written is returned
- if unsuccessful, returns `-1` and set **`errno`**
- does (can) not check **buf** points to **count** bytes of data
- associated with a file descriptor is a ***current position*** in file
- next call to **write** will follow bytes already written
- file often created by repeated calls to write
- can also modify this current position with `lseek`

Hello write!

```
// hello world implemented with libc
#include <unistd.h>
int main(void) {
    char bytes[13] = "Hello, Zac!\n";
    // write takes 3 arguments:
    //   1) file descriptor, 1 == stdout
    //   2) memory address of first byte to write
    //   3) number of bytes to write
    write(1, bytes, 12); // prints Hello, Zac! on stdout
    return 0;
}
```

source code for hello_libc.c

C library wrapper for close system call

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

- release open file descriptor **fd**
- if successful, return **0**
- if unsuccessful, return **-1** and set **errno**
  - could be unsuccessful if **fd** is not an open file descriptor
  - e.g. if **fd** has already been closed
- number of file descriptors may be limited (maybe to 1024)
  - limited number of file open at any time, so use **close()**

An aside: removing a file e.g. via `rm`

- removes the file's entry from a directory
- but the file (inode and data) persist until
  - all references to the file (inode) from other directories are removed
  - all processes accessing the file `close()` their file descriptor
- after this, the operating system reclaims the space used by the files

Using libc system call wrappers to copy a file

```
int main(int argc, char *argv[]) {
    // copy bytes one at a time from  pathname passed as
    // command-line argument 1 to pathname given as argument 2
    int read_file_descriptor = open(argv[1], O_RDONLY);
    int write_file_descriptor = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

source code for cp_libc_one_byte.c

Using libc system call wrappers to copy a file

```
int write_file_descriptor = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0644);
while (1) {
    char bytes[1];
    ssize_t bytes_read = read(read_file_descriptor, bytes, 1);
    if (bytes_read <= 0) {
        break;
    }
    write(write_file_descriptor, bytes, 1);
}
```

source code for cp_libc_one_byte.c

C library wrapper for lseek system call

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

- change the *current position* in stream indicated by **`fd`**
- **`offset`** is in units of bytes, and can be negative
- **`whence`** can be one of ...
  - `SEEK_SET` — set file position to **`offset`** from start of file
  - `SEEK_CUR` — set file position to **`offset`** from current position
  - `SEEK_END` — set file position to **`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
- for example:

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

stdio.h - C Standard Library I/O Functions

- system calls provide operations to manipulate files.
- libc provides a non-portable low-level API to manipulate files
- `stdio.h` provides a portable higher-level API to manipulate files.
- `stdio.h` is part of standard C library
- available in every C implementation that can do I/O
- `stdio.h` functions are portable, convenient & efficient
- use `stdio.h` functions for file operations unless you have a good reason not to
  - e.g .program with special I/O requirements like a database implementation
- on Unix-like systems they will call open()/read()/write()/...
  - but with buffering for efficiency

stdio.h - fopen()

```
FILE *fopen(const char *pathname, const char *mode)
```

- **`fopen()`** is `stdio.h` equivalent to **`open()`**
- **mode** is string of 1 or more characters including:
  - **r** open text file for reading.
  - **w** open text file for writing truncated to 0 zero length if it exists created if does not exist
  - **a** open text file for writing writes append to it if it exists created if does not exist
- fopen returns a **FILE *** pointer
  - **FILE** is `stdio.h` equivalent to file descriptors
  - **FILE** is an opaque struct - we can not access fields
  - **FILE** stores file descriptor
  - **FILE** may also for efficiency store buffered data,

stdio.h - fclose()

```
int fclose(FILE *stream)
```

- **fclose()** is stdio.h equivalent to **close()**
- call **fclose()** as soon as finished with stream
- number of streams open at any time is limited (to maybe 1024)
- stdio functions for efficiency may delay calling **write()**
  - only calls **write()** when it has enough data (perhaps 4096 bytes)
  - also calls **write()** if needed when program exits or **fclose()**
- so last data may not be written until **fclose** or program exit
  - good practice to call **fclose** as soon as finished using stream
- **fflush(stream)** forces any buffered data to be written

stdio.h - read and writing

```
int fgetc(FILE *stream)                           // read a byte
int fputc(int c, FILE *stream)                    // write a byte

char *fputs(char *s, FILE *stream)                // write a string
char *fgets(char *s, int size, FILE *stream)      // read a line

int fscanf(FILE *stream, const char *format, ...)  // formatted input
int fprintf(FILE *stream, const char *format, ...) // formatted output

// read/write array of bytes (fgetc/fputc + loop often better)
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- fputs/fgets, fscanf/fprintf can not be used for binary data because may contain zero bytes
  - can use text (ASCII/Unicode) but can not use to e.g. read a *jpg*
- scanf/fscanf/sscanf often avoided in serious code
  - but fine while learning to code

stdio.h - convenience functions for stdin/stdout

- as we often read/write to stdin/stdout **stdio.h** provides convenience functions, we can use:

```
int getchar()               // fgetc(stdin)
int putchar(int c)          // fputc(c, stdout)

int puts(char *s)           // fputs(s, stdout)

int scanf(char *format, ...)  // fscanf(stdin, format, ...)
int printf(char *format, ...) // fprintf(stdout, format, ...)

char *gets(char *s);        // NEVER USE - major security vulnerability
                            // string may overflow array

// also NEVER USE %s with scanf - similarly major security vulnerability
scanf("%s", array);
```

stdio.h - using fputc to output bytes

```c
char bytes[] = "Hello, stdio!\n"; // 15 bytes
// write 14 bytes so we don't write (terminating) 0 byte
for (int i = 0; i < (sizeof bytes) - 1; i++) {
    fputc(bytes[i], stdout);
}
// or as we know bytes is 0-terminated
for (int i = 0; bytes[i] != '\0'; i++) {
    fputc(bytes[i], stdout);
}
// or if you prefer pointers
for (char *p = &bytes[0]; *p != '\0'; p++) {
    fputc(*p, stdout);
}
```

source code for hello_stdio.c

stdio.h - using fputs, fwrite & fprintf to output bytes

```c
char bytes[] = "Hello, stdio!\n"; // 15 bytes
```

```c
// fputs relies on bytes being 0-terminated
fputs(bytes, stdout);
// write 14 1 byte items
fwrite(bytes, 1, (sizeof bytes) - 1, stdout);
// %s relies on bytes being 0-terminated
fprintf(stdout, "%s", bytes);
```

source code for hello_stdio.c

stdio.h - creating a file

```c
// create file "hello.txt" containing 1 line: Hello, Zac!
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    FILE *output_stream = fopen("hello.txt", "w");
    if (output_stream == NULL) {
        perror("hello.txt");
        return 1;
    }
    fprintf(output_stream, "Hello, Zac!\n");
    // fclose will flush data to file, best to close file ASAP
    // optional here as fclose occurs automatically on exit
    fclose(output_stream);
    return 0;
}
```

source code for create_file_fopen.c

stdio.h - using fgetc to copy a file

```
FILE *input_stream = fopen(argv[1], "r");
if (input_stream == NULL) {
    perror(argv[1]);  // prints why the open failed
    return 1;
}
FILE *output_stream = fopen(argv[2], "w");
if (output_stream == NULL) {
    perror(argv[2]);
    return 1;
}
int c; // not char!
while ((c = fgetc(input_stream)) != EOF) {
    fputc(c, output_stream);
}
fclose(input_stream);  // optional here as fclose occurs
fclose(output_stream); // automatically on exit
```

source code for cp_fgetc.c

I/O Performance & Buffering - libc

```
$ clang -O3 cp_libc_one_byte.c -o cp_libc_one_byte
$ 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_libc_one_byte random_file random_file_copy
real    1m2.878s
user    0m2.248s
sys 1m0.555s
```

I/O Performance & Buffering - stdio buffering

- assume stdio buffering size (`BUFSIZ`) is `4096` (typical)
- first **`fgetc()`** calls requests 4096 bytes via **`read()`**
    - returns 1 byte stores remaining 4095 bytes in an array, the ***input buffer***
- next 4095 **`fgetc()`** calls return a byte from (*input buffer*) and do not to call **`read()`**
- 4097th **`fgetc()`** call requests 4096 bytes via **`read()`**
- returns 1 byte, stores remaining 4095 bytes in the (*input buffer*)
- and so on
- first 4095 **`fputc()`** calls put bytes in an array, the (***output buffer***)
- 4096th **`fputc()`** calls **`write()`** for all 4096 bytes in the ***output buffer***
- and so on
- *output buffer** emptied by **`exit`** or `main` returning
- program can explicitly force empty of output buffer with **`fflush()`** call
- main reason - system calls are expensive

I/O Performance & Buffering - stdio

```
$ clang -O3 cp_fgetc.c -o cp_fgetc
$ time ./cp_fgetc random_file random_file_copy
real    0m0.160s
user    0m0.063s
sys 0m0.027s
```

- at the user level copies 1 byte at time using `fgetc`/`fputc`
- much faster than coping 1 byte at time using `read`/`write`

Copying Multiple Bytes Per Time with System Calls

```
$ clang -O3 cp_libc.c -o cp_libc
$ time ./cp_libc random_file random_file_copy
real    0m0.105s
user    0m0.000s
sys 0m0.037s
```

- faster still because we copy 4096 bytes at a time while avoiding the buffering of fputc.

stdio.h - other operations

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

- **fseek()** is stdio equivalent to **`lseek()`**, just like lseek():
- **`offset`** is in units of bytes, and can be negative
- **`whence`** can be one of ...
  - SEEK_SET — set file position to **`offset`** from start of file
  - SEEK_CUR — set file position to **`offset`** from current position
  - SEEK_END — set file position to **`offset`** from end of file
- for example:

```
fseek(stream, 42, SEEK_SET); // move to after 42nd byte in file
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
```

Using fseek to read the last byte then the first byte of a file

```
FILE *input_stream = fopen(argv[1], "rb");
// move to a position 1 byte from end of file
// then read 1 byte
fseek(input_stream, -1, SEEK_END);
printf("last byte of the file is 0x%02x\n", fgetc(input_stream));
// move to a position 0 bytes from start of file
// then read 1 byte
fseek(input_stream, 0, SEEK_SET);
printf("first byte of the file is 0x%02x\n", fgetc(input_stream));
```

source code for fseek.c

- NOTE: important error checking is missing above

Using fseek to read bytes in the middle of a file

```
// move to a position 41 bytes from start of file
// then read 1 byte
fseek(input_stream, 41, SEEK_SET);
printf("42nd byte of the file is 0x%02x\n", fgetc(input_stream));
// move to a position 58 bytes from current position
// then read 1 byte
fseek(input_stream, 58, SEEK_CUR);
printf("100th byte of the file is 0x%02x\n", fgetc(input_stream));
```

source code for fseek.c

- NOTE: important error checking is missing above

Using fseek to change a random file bit

```
FILE *f = fopen(argv[1], "r+");    // open for reading and writing
fseek(f, 0, SEEK_END);             // move to end of file
long n_bytes = ftell(f);           // get number of bytes in file
srandom(time(NULL));               // initialize random number
                                   // generator with current time
long target_byte = random() % n_bytes; // pick a random byte
fseek(f, target_byte, SEEK_SET);  // move to byte
int byte = fgetc(f);              // read byte
int bit = random() % 8;           // pick a random bit
int new_byte = byte ^ (1 << bit); // flip the bit
fseek(f, -1, SEEK_CUR);           // move back to same position
fputc(new_byte, f);               // write the byte
fclose(f);
```

source code for fuzz.c

- random changes to search for errors/vulnerabilities called fuzzing

Using fseek to create a gigantic sparse file (advanced topic)

```c
// Create a 16 terabyte sparse file
// https://en.wikipedia.org/wiki/Sparse_file
// error checking omitted for clarity
#include <stdio.h>
int main(void) {
    FILE *f = fopen("sparse_file.txt", "w");
    fprintf(f, "Hello, Andrew!\n");
    fseek(f, 16L * 1000 * 1000 * 1000 * 1000, SEEK_CUR);
    fprintf(f, "Goodbye, Andrew!\n");
    fclose(f);
    return 0;
}
```

source code for create_gigantic_file.c

- almost all the 16Tb are zeros which the file system doesn't actually store

stdio.h - I/O to strings

`stdio.h` provides useful functions which operate on strings

```c
// sscanf like scanf, but input comes from char array **str**
int sscanf(const char *str, const char *format, ...);

// snprintf is like printf, but output goes to char array str
// handy for creating strings passed to other functions
// size contains size of str
int snprintf(char *str, size_t size, const char *format, ...);

// also sprintf - more convenient -  but can overflow str
// major security vulnerability -  DO NOT USE
int sprintf(char *str, const char *format, ...); // DO NOT USE
```

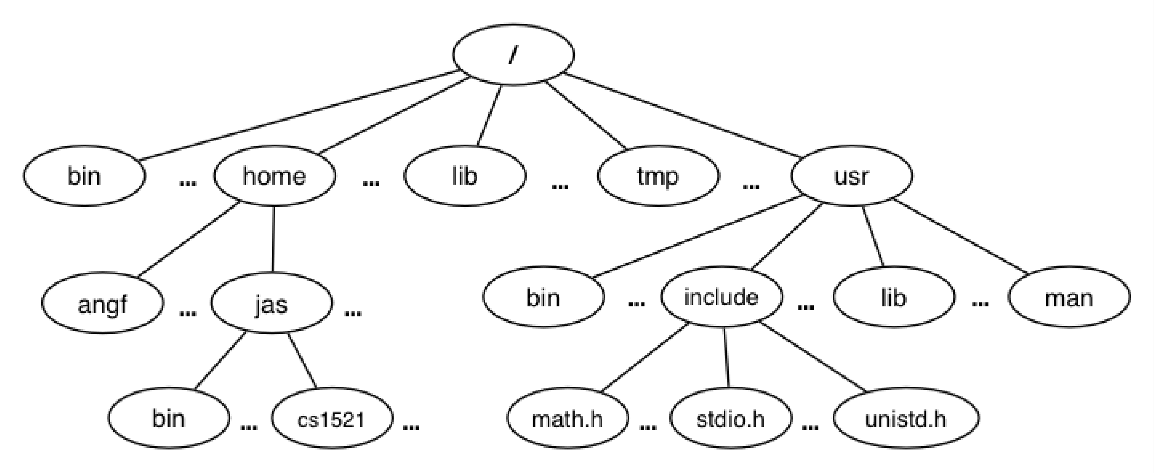
What Really are Files and Directories?

- *file systems* manage persistent stored data e.g. on magnetic disk or SSD
- On Unix-like systems:
    - a *file* is sequence (array) of zero or more bytes.
    - no meaning for bytes associated with file
        - file metadata doesn't record that it is e.g. ASCII, MP4, JPG, …
        - Unix-like files are just bytes
    - a *directory* is an object containing zero or more files or directories.
- file systems maintain metadata for files & directories, e.g. permissions

Unix-like Files & Directories

- Unix-like filenames are sequences of 1 or more bytes.
  - filenames can contain any byte except **`0x00`** and **`0x2F`**
  - **`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 can not be used - they have a special meaning:
  - **`.`** current directory
  - **`..`** parent directory
- Some programs (shell, ls) treat filenames starting with **`.`** specially.
- Unix-like directories are sets of files or directories

Unix/Linux File System



- Unix/Linux file system is tree-like
- Exception: if you follow symbolic links it is a *graph*.
  - and you may infinitely loop attempting to traverse a file system
  - but only if you follow symbolic links

Unix/Linux Pathnames

- Files & directories accessed via pathnames, e.g: `/home/z5555555/lab07/main.c`
- *absolute* pathnames start with a leading **/** and give full path from root
  - e.g. `/usr/include/stdio.h`, `/cs1521/public_html/`
- every process (running program) has a *current working directory* (CWD)
  - this is an absolute pathname
- shell command **`pwd`** prints *current working directory*
- *relative* pathname do not start with a leading **/**
  - e.g. `../../another/path/prog.c`, `./a.out`, `main.c`
- *relative* pathnames appended to *current working directory* of process using them
- Assume process *current working directory* is `/home/z5555555/lab07/`
  - `main.c` translated to absolute path `/home/z5555555/lab07/main.c`
  - `../a.out` translated to absolute path `/home/z5555555/lab07/../a.out`
  - which is equivalent to absolute path `/home/z5555555/a.out`

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
  - ...

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

File system implementations often add complexity to improve performance

- e.g. very small files might be stored in an inode itself

File Inodes

- unix-like file systems effectively have a large array of inodes containg metadata
- an inode's index in this array is its ***inode-number*** (or ***i-number***)
- inode-number uniquely identify files within a filesystem
  - just a zid uniquely identifies a student within UNSW
- directories are effectively a list of (name, inode-number) pairs
- **`ls -i`** prints **inode-numbers**

```
$ ls -i file.c
109988273 file.c
$
```

- note there is usually more than one file systems mounted on a Unix-like system
  - each file-systems has a separate set of **inode-numbers**
  - files on different file-systems could have the same **inode-number**

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”
  - collect information about file’s location and size
  - update access timestamp
- use data in inode 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:

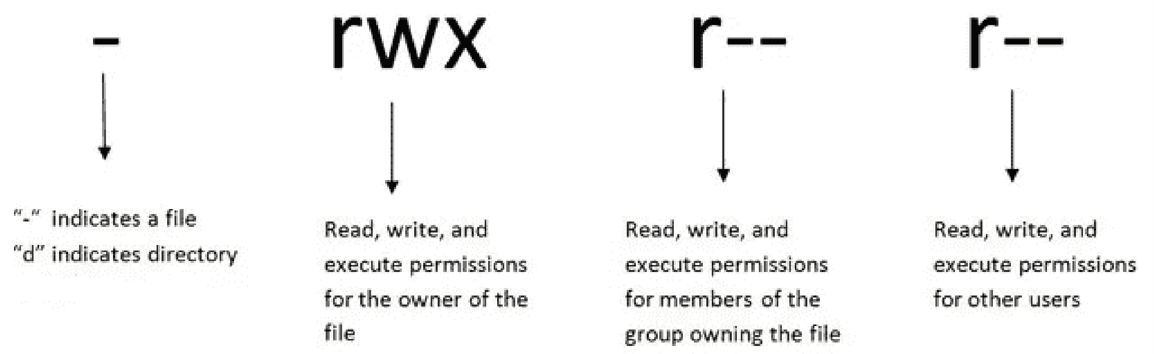
- user: the file’s owner
- group: the members of the file’s group
- other: everyone else

read, write and execute have slightly different meanings for files vs directories:

- read: For a normal file, read permission allows a user to view the contents of the file. For a directory, read permission allows a user to view the names of the items in the directory eg use `ls`
- write: For a normal file, write permission allows a user to modify and delete the file. For a directory, write permission allows files within the directory to be created, deleted or renamed.
- execute: For a normal file, execute permission allows a user to execute a file. For a directory it means a user may enter the directory eg `cd` into it. It is also necessary to be able to access(read, write, execute) items in the directory.

File Permissions

Permissions are broken into 4 sections



- rwx r-- r--

“-” indicates a file
“d” indicates directory

Read, write, and execute permissions for the owner of the file

Read, write, and execute permissions for members of the group owning the file

Read, write, and execute permissions for other users

You can see file permissions in linux by typing

```
$ ls -l
```

File Permissions

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

```
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. eg.

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

Hard Links & 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
  - all hard links to a file have equal status
  - file destroyed when last hard link removed
  - can not create a (extra) hard link to directories
- 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)

Hard Links & Symbolic Links

```
$ echo 'Hello Andrew' >hello
$ ln hello hola        # create hard link
$ ln -s hello selamat # create symbolic link
$ ls -l hello hola selamat
-rw-r--r-- 2 andrewt 13 Oct 23 16:18 hello
-rw-r--r-- 2 andrewt 13 Oct 23 16:18 hola
lrwxrwxrwx 1 andrewt  5 Oct 23 16:20 selamat -> hello
$ cat hello
Hello Andrew
$ cat hola
Hello Andrew
$ cat selamat
Hello Andrew
```

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

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

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

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

- same as `stat()` but doesn't follow symbolic links

definition of struct 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 */
  blksize_t st_blksize;     /* Block size for filesystem I/O */
  blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */
  struct timespec st_atim;  /* Time of last access */
  struct timespec st_mtim;  /* Time of last modification */
  struct timespec st_ctim;  /* Time of last status change */
};
```

st_mode field of struct stat

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

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_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

Using stat

```
struct stat s;
if (stat(pathname, &s) != 0) {
    perror(pathname);
    exit(1);
}
printf("ino =  %10ld # Inode number\n", s.st_ino);
printf("mode = %10o # File mode \n", s.st_mode);
printf("nlink =%10ld # Link count \n", (long)s.st_nlink);
printf("uid =  %10u # Owner uid\n", s.st_uid);
printf("gid =  %10u # Group gid\n", s.st_gid);
printf("size = %10ld # File size (bytes)\n", (long)s.st_size);
printf("mtime =%10ld # Modification time (seconds since 1/1/70)\n",
       (long)s.st_mtime);
```

source code for stat.c

mkdir

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

- create a new directory called **pathname** with permissions **mode**
- if **pathname** is e.g. a/b/c/d
  - all of the directories **a**, **b** and **c** must exist
  - directory **c** must be writeable 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
- returns 0 if successful, returns -1 and sets **errno** otherwise
- for example:

```
    mkdir("newDir", 0755);
```

Example of using mkdir to create directories

```
#include <stdio.h>
#include <sys/stat.h>
// create the directories specified as command-line arguments
int main(int argc, char *argv[]) {
    for (int arg = 1; arg < argc; arg++) {
        if (mkdir(argv[arg], 0755) != 0) {
            perror(argv[arg]);  // prints why the mkdir failed
            return 1;
        }
    }
    return 0;
}
```

source code for mkdir.c

POSIX functions to access directory contents

```c
#include <sys/types.h>
#include <dirent.h>

// 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);
```

Using opendir/readdir to print directory contents

```c
int main(int argc, char *argv[]) {
    for (int arg = 1; arg < argc; arg++) {
        DIR *dirp = opendir(argv[arg]);
        if (dirp == NULL) {
            perror(argv[arg]);  // prints why the open failed
            return 1;
        }
        struct dirent *de;
        while ((de = readdir(dirp)) != NULL) {
            printf("%ld %s\n", de->d_ino, de->d_name);
        }
        closedir(dirp);
    }
```

source code for list_directory.c

Other useful Linux (POSIX) functions

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

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

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
```

file permissions

- file permissions are separated into three types:
  - **read** - permission to get bytes of file
  - **write** - permission to change bytes of file
  - **execute** - permission to execute file
- read/write/execute often represented as bits of an octal digit
- file permissions are specified for 3 groups of users:
  - **owner** - permissions for the file owner
  - **group** - permissions for users in the group of the file
  - **other** - permissions for any other user

changing file permissions

```
// first argument is mode in octal
mode_t mode = strtol(argv[1], &end, 8);
// check first argument was a valid octal number
if (argv[1][0] == '\0' || end[0] != '\0') {
    fprintf(stderr, "%s: invalid mode: %s\n", argv[0], argv[1]);
    return 1;
}
for (int arg = 2; arg < argc; arg++) {
    if (chmod(argv[arg], mode) != 0) {
        perror(argv[arg]);  // prints why the chmod failed
        return 1;
    }
}
```

source code for chmod.c

removing files

```
// remove the specified files
int main(int argc, char *argv[]) {
    for (int arg = 1; arg < argc; arg++) {
        if (unlink(argv[arg]) != 0) {
            perror(argv[arg]);  // prints why the unlink failed
            return 1;
        }
    }
    return 0;
}
```

source code for rm.c

```
$ dcc rm.c
$ ./a.out rm.c
$ ls -l rm.c
ls: cannot access 'rm.c': No such file or directory
```

renaming a file

```c
// rename the specified file
int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <old-filename> <new-filename>\n",
                argv[0]);
        return 1;
    }
    char *old_filename = argv[1];
    char *new_filename = argv[2];
    if (rename(old_filename, new_filename) != 0) {
        fprintf(stderr, "%s rename %s %s:", argv[0], old_filename,
                new_filename);
        perror("");
        return 1;
    }
    return 0;
}
```

cd-ing up one directory at a time

```c
// use repeated chdir("..") to climb to root of the file system
char pathname[PATH_MAX];
while (1) {
    if (getcwd(pathname, sizeof pathname) == NULL) {
        perror("getcwd");
        return 1;
    }
    printf("getcwd() returned %s\n", pathname);
    if (strcmp(pathname, "/") == 0) {
        return 0;
    }
    if (chdir("..") != 0) {
        perror("chdir");
        return 1;
    }
}
```

source code for getcwd.c

making a 1000-deep directory (advanced)

```c
for (int i = 0; i < 1000;i++) {
    char dirname[256];
    snprintf(dirname, sizeof dirname, "d%d", i);
    if (mkdir(dirname, 0755) != 0) {
        perror(dirname);
        return 1;
    }
    if (chdir(dirname) != 0) {
        perror(dirname);
        return 1;
    }
    char pathname[1000000];
    if (getcwd(pathname, sizeof pathname) == NULL) {
        perror("getcwd");
        return 1;
    }
    printf("\nCurrent directory now: %s\n", pathname);
}
```

source code for nest_directories.c

creating 1000 hard links to a file - creating the file (advanced)

```c
int main(int argc, char *argv[]) {
    char pathname[256] = "hello.txt";
    // create a target file
    FILE *f1;
    if ((f1 = fopen(pathname, "w")) == NULL) {
        perror(pathname);
        return 1;
    }
    fprintf(f1, "Hello Andrew!\n");
    fclose(f1);
```

source code for many_links.c

creating 1000 hard links to a file - checking the file (advanced)

```c
for (int i = 0; i < 1000; i++) {
    printf("Verifying '%s' contains: ", pathname);
    FILE *f2;
    if ((f2 = fopen(pathname, "r")) == NULL) {
        perror(pathname);
        return 1;
    }
    int c;
    while ((c = fgetc(f2)) != EOF) {
        fputc(c, stdout);
    }
    fclose(f2);
```

source code for many_links.c

creating 1000 hard links to a file (creating a link)

```c
        char new_pathname[256];
        snprintf(new_pathname, sizeof new_pathname,
                 "hello_%d.txt", i);
        printf("Creating a link %s -> %s\n",
               new_pathname, pathname);
        if (link(pathname, new_pathname) != 0) {
            perror(pathname);
            return 1;
        }
    }
    return 0;
}
```

source code for many_links.c