Operating system - What Does it Do.

- OS sits between the user and the hardware
- OS provides effectively a virtual machine to user
- much simpler and more convenient than real machine
- interface can be consistent across different hardware
- can coordinate/share access to resources between users
- can provide privileges/security

Operating Systems - What Does it Need from Hardware.

- needs hardware to provide a **privileged** mode which:
  - allows access to all hardware/memory
  - Operating System (kernel) runs in **privileged** mode
  - allows transfer to running code a **non-privileged** mode
- needs hardware to provide a **non-privileged** mode which:
  - prevents access to hardware
  - limits access to memory
  - provides mechanism to make requests to operating system
- operating system request called a system call
  - transfers execution back to kernal code in **privileged** mode

System Call - What is It

- system call transfers execution to **privileged** mode and executes operating code
- includes arguments specifying details of request being made
- Linux provides 400+ system calls
- Examples:
  - get bytes from a file
  - request more memory
  - create a process (run a program)
  - terminate a process
  - send or receive information via a network

SPIM

- SPIM provides a virtual machine which can execute MIPS programs
- SPIM also provides a tiny operating system
- small number of SPIM system calls for I/O and memory allocation
- access is via the **`syscall`** instruction
- MIPS programs running on real hardware + real OS (linux) also use syscall instruction

SPIM System Calls

Service	$v0	Arguments	Result
printf("%d")	1	int in $a0	-
printf("%f")	2	float in $f12	-
printf("%lf")	3	double in $f12	-
printf("%s")	4	$a0 = string	-
scanf("%d")	5	-	int in $v0
scanf("%f")	6	-	float in $f0
scanf("%lf")	7	-	double in $f0
fgets	8	buffer address in $a0	
		length in $a1	-
sbrk	9	nbytes in $a0	address in $v0
printf("%c")	11	char in $a0	-
scanf("%c")	12	-	char in $v0
exit(status)	17	status in $a0	-

Files and Directories

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

On Unix-like systems:

- a *file* is sequence (array) of zero or more bytes.
- and a *directory* is an object containing zero or more files or directories.
- file system maintains metadata for files & directories , e.g. permissions
- system calls provide operations to manipulate files.
- libc provides low-level API to manipulate files.
- `stdio.h` provides more portable, higher-level API to manipulate files.

Unix/Linux Pathnames

- Files & directories accessed via pathnames, e.g:
  `/home/z5555555/lab07/main.c`
- Unix pathnames is a sequence of any byte.
- Except filenames can not contain 0 ('\0') bytes.
  - because pathnames stored in null-terminated strings
- And filenames can not contain ASCII '/' (0x2F)
  - because '/' used to separate components of path.
- Also 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/Linux Pathnames

- *absolute* pathnames start with a leading /
- *absolute* pathnames give full path from root

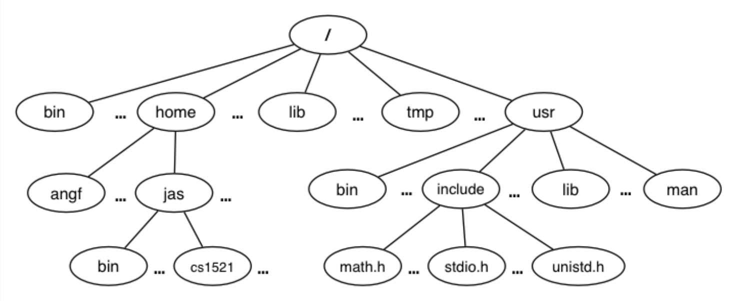
  e.g. `/usr/include/stdio.h`, `/cs1521/public_html/`
- every process (running process) has an associated *absolute* pathname called the *current working directory* (CWD)
- shell command `pwd` prints CWD
- *relative* pathname do not start with a leading / e.g. `../../another/path/prog.c`, `./a.out`, `main.c`
- *relative* pathnames appended to CWD of process using them
- Assume process CWD is `/home/z5555555/lab07/`
  `main.c` translated to `/home/z5555555/lab07/main.c`
  `../a.out` translated to `/home/z5555555/../a.out`
  which is equivalent to `/home/z5555555/a.out`

Everything is a File

- Originally file systems managed data stored on a magnetic disk.
- Unix philosophy is: *Everything is a File*.
- File system can be used to access:
  - files
  - directories (folders)
  - storage devices (disks, SSD, ...)
  - peripherals (keyboard, mouse, USB, ...)
  - system information
  - inter-process communication
  - ...

Unix/Linux File System

Unix/Linux file system is tree-like



We think of file-system as a *tree* but links actually make it a *graph*.

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)
- ownership, access permissions
- timestamps (create/access/update)

Note: file systems add much complexity to improve performance

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

File Inodes

- file systems effectively have an array of inodes
- index in this array is inode's unique i-number
- directories are effectively a list of (name,i-number) pairs
- i-numbers uniquely identify within filesystem (like UNSW zid)
- **`ls -i`** prints i-number, e.g.:

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

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*,`ino`), access inode table `inodes[ino]`
- 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 indoe to access file contents

Hard Links & Symbolic Links

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

Hard links

- multiple directory entries referencing the same file (inode)
- the two entries must be on the same filesystem

Symbolic links (symlinks)

- a file containing the path name of another file
- opening the symlink opens the file being referenced

Hard Links & Symbolic Links

```
$ echo 'Hello Andrew' >hello
$ ln hello hola        # create hard link
$ ln -s hello selamat
$ 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
```

File Operations: Overview

Unix presents a uniform interface to file system objects

- functions/syscalls manipulate objects as a *stream of bytes*
- accessed via a *file descriptor*
    - file descriptor index into a per-process operating system table

Some common operations:

- `open()` — open a file system object, returning a file descriptor
- `close()` — stop using a file descriptor
- `read()` — read some bytes into a buffer from a file descriptor
- `write()` — write some bytes from a buffer to a file descriptor
- `lseek()` — move to a specified offset within a file
- `stat()` — get meta-data about a file system object

Extra Types for File System Operations

Unix defines a range of file-system-related types:

- **off_t** — offsets within files
  - typically **int64_t** - signed to allow backward refs
- **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
  - like `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`, ...

open

```
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`

close

```c
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

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

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

read

```
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
- next call to **read** will return next bytes from file
- repeated calls to reads will yield entire contents of file
  - associated with a file descriptor is "current position" in file
  - can also modify this position with `lseek`

write

```
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, return -1 and set `errno`
- does (can) not check **buf** points to **count** bytes of data
- next call to **write** will follow bytes already written
- file often created by repeated calls to write
  - associated with a file descriptor is "current position" in file
  - can also modify this position with `lseek`

lseek

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

- change the 'current position' in the file of **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

Example: `lseek(fd, 0, SEEK_END);` (move to end of file)

stat

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

- stores meta-data associated with **pathname** into **statbuf**
- information includes
    - inode number, file type + access mode, owner, group
    - size in bytes, storage block size, allocated blocks
    - time of last access/modification/status-change
- returns -1 and sets `errno` if meta-data 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

stat st_mode

The `st_mode` is a bit-string containing some of:

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

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

Example:

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

Other useful Linux (POSIX) functions

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

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

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

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

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

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

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

stdio.h

`stdio.h` functions more portable more convenient than open/read/write/... use them by default

- `stdio.h` equivalent to open is **fopen**

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

- **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 an opaque struct - we can not access fields

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

- `stdio.h` equivalent to `close`

stdio - 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

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

// read array of bytes
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
// write array of bytes
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream)
```

stdio.h - convenience functions for stdin/stdout

As we often read/write to stdin/stdout `stdio.h` provides convenience functions which only read/write stdin/stdout

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

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
```

stdio.h - other operations on

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

- **fseek** is stdio equivalent to lseek
- like `lseek` **offset** can be postive or negative
- like `lseek` **whence** can be `SEEK_SET`, `SEEK_CUR` or `SEEK_END` making **offset** relavtive to file start, current position or file end

```
int fflush(FILE *stream);
```

- flush any buffered data on writing stream

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

- equivalent to `close`

stdio.h - I/O to strings

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

```
int snprintf(char *str, size_t size, const char *format, ...);
```

- like printf, but output goes to char array **str**
- handy for creating strings passed to other functions
- do not use unsafe related function: ‘sprintf

```
int sscanf(const char *str, const char *format, ...);
```

- like scanf, but input comes from char array **str**

```
int sprintf(char *str, const char *format, ...); // DO NOT USE
```

- like **snprintf** but dangerous because can overflow **str**

File System Summary

Operating systems provide a *file system*

- as an abstraction over physical storage devices (e.g. disks)
- providing named access to chunks of related data (files)
- providing access (sequential/random) to the contents of files
- allowing files to be arranged in a hierarchy of directories
- providing control over access to files and directories
- managing other meta-data associated with files (size, location, . . .)

Operating systems also manage other resources

- memory, processes, processor time, i/o devices, networking, . . .