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
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 kernel 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

<table>
<thead>
<tr>
<th>Service</th>
<th>$v0</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>printf(“%d”)</td>
<td>1</td>
<td>int in $a0</td>
<td>-</td>
</tr>
<tr>
<td>printf(“%f”)</td>
<td>2</td>
<td>float in $f12</td>
<td>-</td>
</tr>
<tr>
<td>printf(“%lf”)</td>
<td>3</td>
<td>double in $f12</td>
<td>-</td>
</tr>
<tr>
<td>printf(“%s”)</td>
<td>4</td>
<td>$a0 = string</td>
<td>-</td>
</tr>
<tr>
<td>scanf(“%d”)</td>
<td>5</td>
<td>-</td>
<td>int in $v0</td>
</tr>
<tr>
<td>scanf(“%f”)</td>
<td>6</td>
<td>-</td>
<td>float in $f0</td>
</tr>
<tr>
<td>scanf(“%lf”)</td>
<td>7</td>
<td>-</td>
<td>double in $f0</td>
</tr>
<tr>
<td>fgets</td>
<td>8</td>
<td>buffer address in $a0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>length in $a1</td>
<td></td>
</tr>
<tr>
<td>sbrk</td>
<td>9</td>
<td>nbytes in $a0</td>
<td>address in $v0</td>
</tr>
<tr>
<td>printf(“%c”)</td>
<td>11</td>
<td>char in $a0</td>
<td>-</td>
</tr>
<tr>
<td>scanf(“%c”)</td>
<td>12</td>
<td>-</td>
<td>char in $v0</td>
</tr>
<tr>
<td>exit(status)</td>
<td>17</td>
<td>status in $a0</td>
<td>-</td>
</tr>
</tbody>
</table>
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 is tree-like

We think of file-system as a tree but links actually make it a graph.
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
$ 
```
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
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
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
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`, ...
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**
int close(int fd)

- release open file descriptor \texttt{fd}
- if successful, return 0
- if unsuccessful, return -1 and set \texttt{errno}
  - could be unsuccessful if \texttt{fd} is not an open file descriptor
    e.g. if \texttt{fd} has already been closed

An aside: removing a file e.g. via \texttt{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 \texttt{close()} their file descriptor
- after this, the inode and the space used for file contents is recycled
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
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**
off_t lseek(int fd, off_t offset, int whence)

- change the ‘current position’ in the file of \texttt{fd}
- \texttt{offset} is in units of bytes, and can be negative
- \texttt{whence} can be one of …
  - \texttt{SEEK_SET} — set file position to \textit{Offset} from start of file
  - \texttt{SEEK_CUR} — set file position to \textit{Offset} from current position
  - \texttt{SEEK_END} — set file position to \textit{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: \texttt{lseek(fd, 0, SEEK_END)}; (move to end of file)
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
The `st_mode` is a bit-string containing some of:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IFLNK</td>
<td>0120000</td>
<td>symbolic link</td>
</tr>
<tr>
<td>S_IFREG</td>
<td>0100000</td>
<td>regular file</td>
</tr>
<tr>
<td>S_IFBLK</td>
<td>0060000</td>
<td>block device</td>
</tr>
<tr>
<td>S_IFDIR</td>
<td>0040000</td>
<td>directory</td>
</tr>
<tr>
<td>S_IFCHR</td>
<td>0020000</td>
<td>character device</td>
</tr>
<tr>
<td>S_IFIFO</td>
<td>0010000</td>
<td>FIFO</td>
</tr>
<tr>
<td>S_IRUSR</td>
<td>0000400</td>
<td>owner has read permission</td>
</tr>
<tr>
<td>S_IWUSR</td>
<td>0000200</td>
<td>owner has write permission</td>
</tr>
<tr>
<td>S_IXUSR</td>
<td>0000100</td>
<td>owner has execute permission</td>
</tr>
<tr>
<td>S_IRGRP</td>
<td>0000040</td>
<td>group has read permission</td>
</tr>
<tr>
<td>S_IWGRP</td>
<td>0000020</td>
<td>group has write permission</td>
</tr>
<tr>
<td>S_IXGRP</td>
<td>0000010</td>
<td>group has execute permission</td>
</tr>
<tr>
<td>S_IROTH</td>
<td>0000004</td>
<td>others have read permission</td>
</tr>
<tr>
<td>S_IWOTH</td>
<td>0000002</td>
<td>others have write permission</td>
</tr>
<tr>
<td>S_IXOTH</td>
<td>0000001</td>
<td>others have execute permission</td>
</tr>
</tbody>
</table>
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:

```c
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 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

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

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

- **fseek** is stdio equivalent to lseek
- like lseek **offset** can be positive or negative
- like lseek **whence** can be SEEK_SET, SEEK_CUR or SEEK_END
  making **offset** relatvive 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

```c
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

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

```c
int sprintf(char *str, const char *format, ...); // DO NOT USE
```
- like `snprintf` but dangerous because can overflow `str`
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, ...