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 **non-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	-

Everything is a File

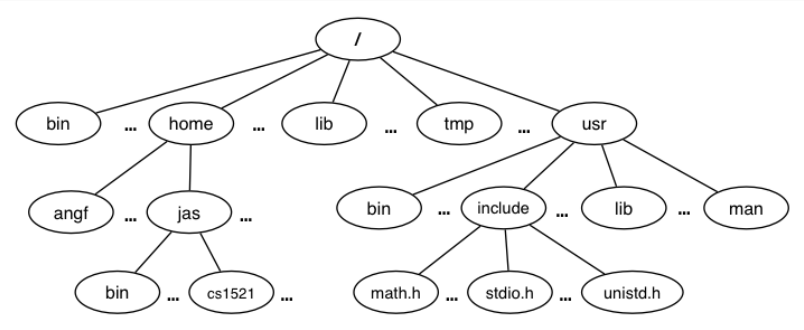
- Originally file systems managed data stored on a magnetic disk.
- Unix philosophy is: *Everything is a File*.
- File system can be used to access:
  - files
  - directories (folders)
  - storage devices (disks, SSD, ...)
  - peripherals (keyboard, mouse, USB, ...)
  - system information
  - inter-process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

Unix/Linux Pathnames

- Files & directories accessed via pathnames, e.g: `/home/z5555555/lab07/main.c`
- Unix pathnames is a sequence of any byte.
- Except pathnames can not contain 0 ('\0') bytes.
- use null-terminated strings for pathnames.
- ASCII '/' (0x2F) used to separate components of path.
- Hence '/' (0x2F) not allowed elsewhere in pathnames
- Two names can not be used - they have a special meaning:
  - . current directory
  - .. parent directory
- Some programs (shell, ls) treat filenames starting with '.' specially.

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*.

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`

Files and Directories

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

- Ordinary *file* sequence (array) of zero or more bytes.
- file system maintains meta-data (e.g., access rights)
- System calls provide low-level API to manipulate files.
- `stdio.h` provides more portable, higher-level API to manipulate files.
- *directory* is an object containing zero or more files or directories.

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 `ino_t`, `dev_t`, `time_t`, `uid_t`, ...

File Metadata

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

- physical location on storage device of file data
- file type (regular file, directory, ...), file size (bytes/blocks)
- ownership, access permissions, timestamps (create/access/update)

Each file system *volume* has a table of inodes in a known location

Note: an inode does not contain the name of the file

Access to a file by name requires a *directory*

- where a directory is effectively a list of (name,inode) pairs

Note: very small files can potentially be stored in an inode (inlining)

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 physical location to access device and manipulate file data

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* (index into a 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

open

`int open(char *`*Path*`, int` *Flags*`)`

- attempt to open an object at *Path*, according to *Flags*
- flags (defined in `<fcntl.h>>`)
  - `O_RDONLY` — open object for reading
  - `O_WRONLY` — open object for writing
  - `O_APPEND` — append on each write
  - `O_RDWR` — open object for reading and writing
  - `O_CREAT` — create object 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 +ve `int`)
- if unsuccessful, return -1 and set `errno`

close

`int close(int`*FileDesc*`)`

- attempt to release an open file descriptor
- if this is the last reference to object, release its resources
- if successful, return 0
- if unsuccessful, return -1 and set `errno`

Could be unsuccessful if *FileDesc* is not an open file descriptor

An aside: removing an object e.g. via `rm`

- removes the object's entry from a directory
- but the inode and data persist until
  - all processes accessing the object `close()` their handle
  - all references to the inode from other directories are removed
- after this, the inode and the blocks on storage device are recycled

read

`ssize_t read(int` *FileDesc*`, void *`*Buffer*`, size_t` *Count*`)`

- attempt to read *Count* bytes from *FileDesc* into *Buffer*
- if 'successful', return number of bytes actually read (*NRead*)
- if currently positioned at end of file, return 0
- if unsuccessful, return -1 and set `errno`
- does not check whether *Buffer* contains enough space
- advances the file offset by *NRead*
- does not treat `' n'` as special, nor is there `EOF`

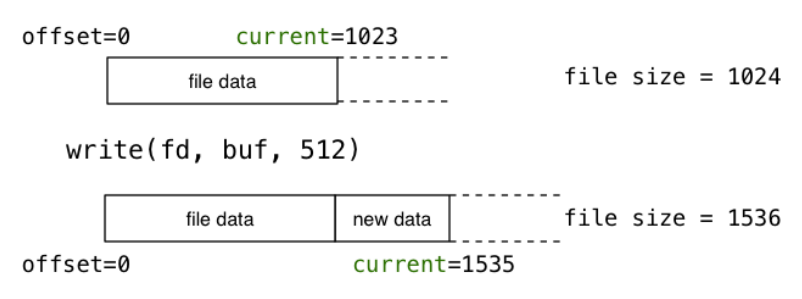
Once a file is `open()`'d ...

- the "current position" in the file is maintained as part of the fd entry
- the "current position" is modified by `read()`, `write()` and `lseek()`

write

`ssize_t write(int` *FileDesc*`, void *`*Buffer*`, size_t` *Count*`)`

- attempt to write *Count* bytes from *Buffer* onto *FileDesc*
- if 'successful', return number of bytes actually written (*NWritten*)
- if unsuccessful, return -1 and set `errno`
- does not check whether *Buffer* has *Count* bytes of data
- advances the file offset by *NWritten* bytes



lseek

`off_t lseek(int` *FileDesc*`, off_t` *Offset*`, int` *Whence*`)`

- set the 'current position' of the *FileDesc*
- *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)

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
```


stat

`int stat(char *FileName, struct stat *StatBuf)`

- stores meta-data associated with *FileName* 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(intFileDesc, struct stat *StatBuf)`

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

`int lstat(char *FileName, 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(char *PathName, mode_t Mode)`

- create a new directory called *PathName* with mode *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 a hard link to a file/directory/...
- symlink(char *target, char *linkpath) — create a symbolic link to a file/directory/...
- unlink(char *pathname) — remove a file/directory/...
- chmod(char *pathname, mode_t mode) — change permission of file/directory/...

stdio.h

The stdio.h provides is more portable and more convenient than open/read/write/... Use it instead when possible.

- `FILE *` — handle on an open file (and a buffer)
- `FILE *fopen(`*Name*`,`*Mode*`)` (*Mode* e.g. `"r"`, `"w"`, `"a"`)
- `int fclose(FILE *`*Stream*`)` (*Stream* from `fopen()`)
- `char *fgets(char *`*Buffer*`, int` *Size*`, FILE *`*Stream*`)`
- `char *fputs(char *`*Buffer*`, FILE *`*Stream*`)`
- `int fscanf(FILE *`*Stream*`, char *`*Format*`, ...)`
- `int fprintf(FILE *`*Stream*`, char *`*Format*`, ...)`
- `int fgetc(FILE *`*Stream*`)`
- `int fputc(int`*Character*`, FILE *`*Stream*`)`

Also, specialised versions of I/O functions, e.g.

- scanf(...) == fscanf(stdin, ...)

stdio.h

The stdio.h also provides equivalent function which operate on strings

- "snprintf(char *str, size_t size, char *format, ...);
  - like printf, but output goes to char array
  - handy for creating strings passed to other functions
  - do not use unsafe related function: sprintf
- "sscanf(const char *str, char *format, ...);
  - like scanf, but input comes from char array

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, ...