Announcements

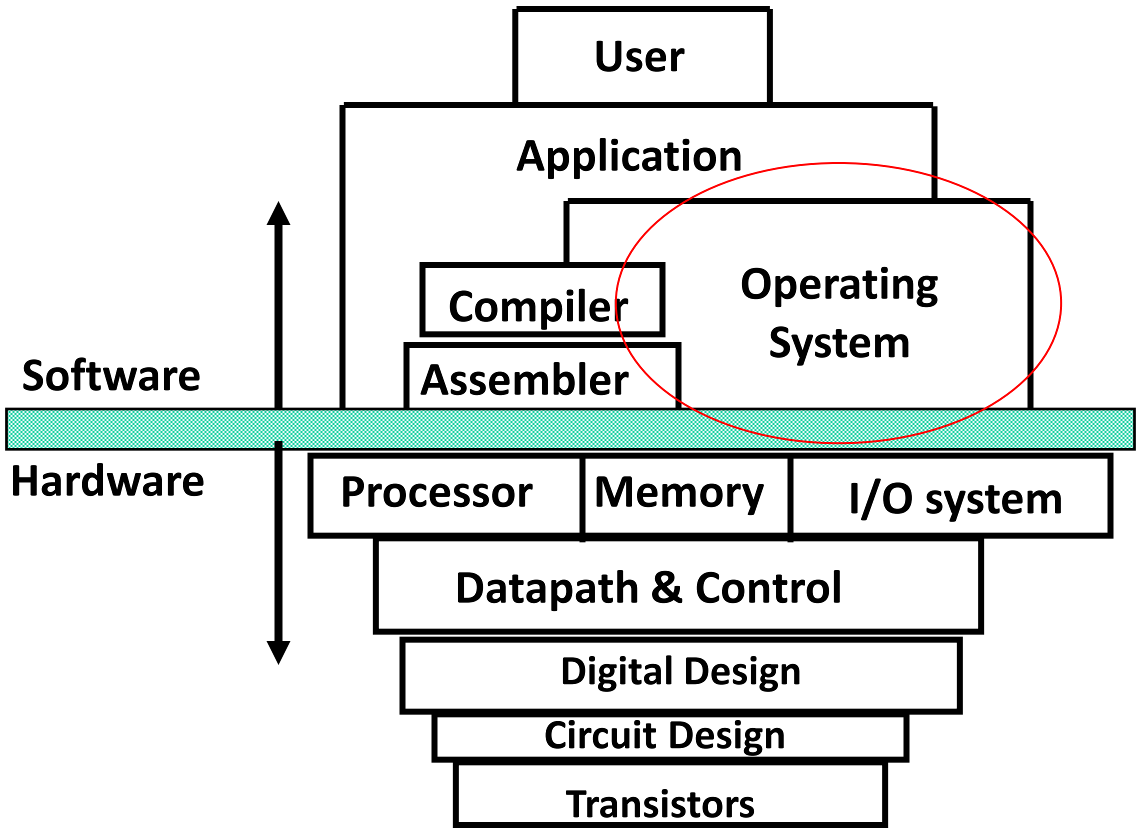
- Exam 3
  - Friday, November 21
  - Covers up to Cache: Lectures 19-28
  - Format: multiple-choice (30%) / short-answer (70%)
- Assignment 8
  - Posted Thursday, November 20; due December 2
- Quiz 11
  - Monday, December 1
  - Covers Lectures 29-31
- Graded Lab 3
  - December 3-4
  - Covers Assignments 4-7

Lecture 30

Systems Programming

CPSC 275
Introduction to Computer Systems

A typical computer system



The Linux System

- *Kernel* – heart of the OS (always running)
  - Process scheduling
  - Memory management
  - I/O control
  - many more …
- *Shell* – Interpreter between the user and the computer
  - e.g., `bash`
- Tools and applications
  - Accessible from shell
  - Can be run independently of shell

Shell

- Provides command line as an interface between the user and the system
- Starts automatically when you log in

  e.g., `bash` for Ubuntu Linux
- Uses a command *language*
  - Allows programming (shell scripting) within the shell environment
    - Uses variables, loops, conditionals, etc.
  - Accepts commands and often makes *system calls* to carry them out

What is Systems Programming?

- Application Programming
  - Solve a user-facing problem (e.g., web browser, text editor).
  - High-level (e.g., Python, Java, JavaScript). Hides machine details.
  - User interface and features.
- Systems Programming
  - Provide services for other software to run (e.g., OS, drivers, compilers, shells).
  - Low-level (e.g., C, C++, Rust). Directly manages system resources.
  - Resource management, performance, efficiency, concurrency.

User Space vs. Kernel Space

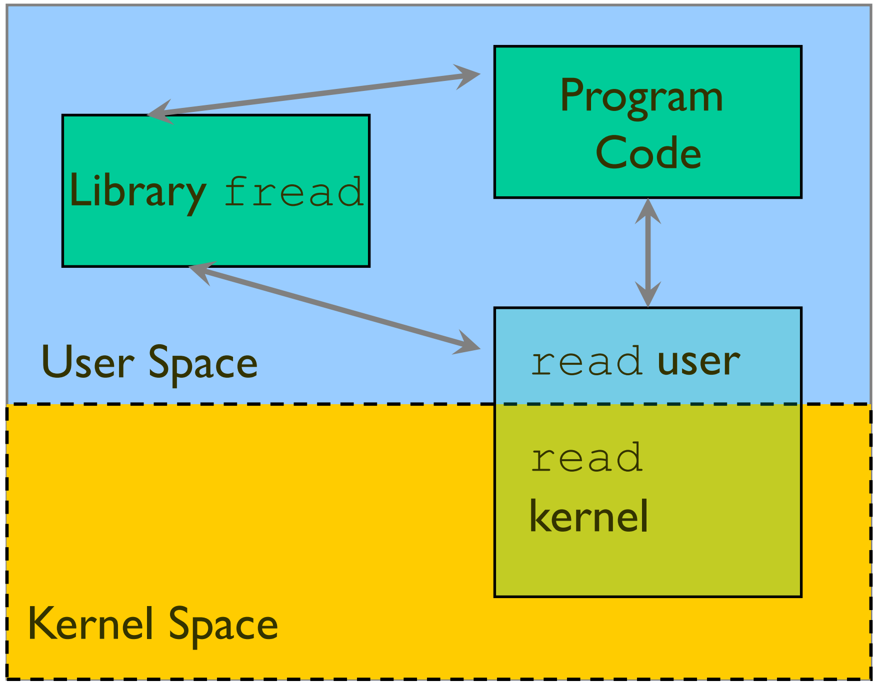
- Kernel space (or kernel mode):
  - Where the Operating System (OS) kernel runs.
  - *Privileged*: Has full, unrestricted access to all hardware
  - If it crashes, the whole system crashes.
- User space (or user mode):
  - Where your applications run (e.g., bash, Chrome, your C program).
  - *Unprivileged*: Cannot directly access hardware.
  - If it crashes, the OS cleans it up. The system survives.
  - Lives in a "container" of its own virtual memory.

Q: If user mode is unprivileged, how does it print to the screen?

System Calls

A: Must ask the kernel (OS) to do it.

- A *system call* is a formal request from a user-space program for a kernel-level service.
  - User program hands over a request by making a system call,
  - Kernel goes into the back, does the privileged work, and
  - Returns the result to the user program.



System calls

- Types of system calls
  - Process management
  - Memory management
  - File I/O
  - Inter-process communication Signal handling
  - …

- Examples:
  - `open()`: Ask the kernel to open a file.
  - `read()`, `write()`: Ask the kernel to move data.
  - `fork()`: Ask the kernel to create a new process.
  - many more ...

Linux filesystem

- The filesystem is your interface to
  - physical storage (disks) on your machine
  - storage on other machines
  - output devices
  - etc.
- *Everything* in Linux is a file (programs, text, peripheral devices, terminals, …)
- Provides a *logical* view of the storage devices

Working directory

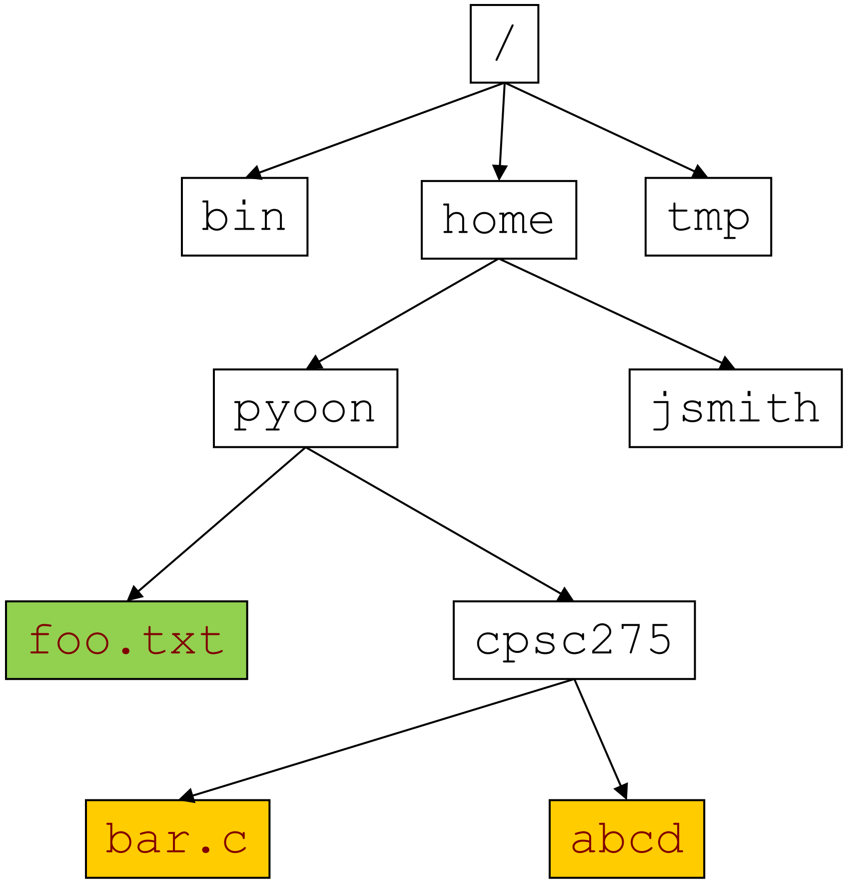
- The current directory in which you are working
- `pwd` command: outputs the absolute path (more on this later) of your working directory
- Unless you specify another directory, commands will assume you want to operate on the working directory

Home directory

- A special place for each user to store personal files
- When you log in, your working directory will be set to your home directory
- Your home directory is represented by the symbol ~ (tilde)
    - The home directory of "`user`" is represented by `~user`

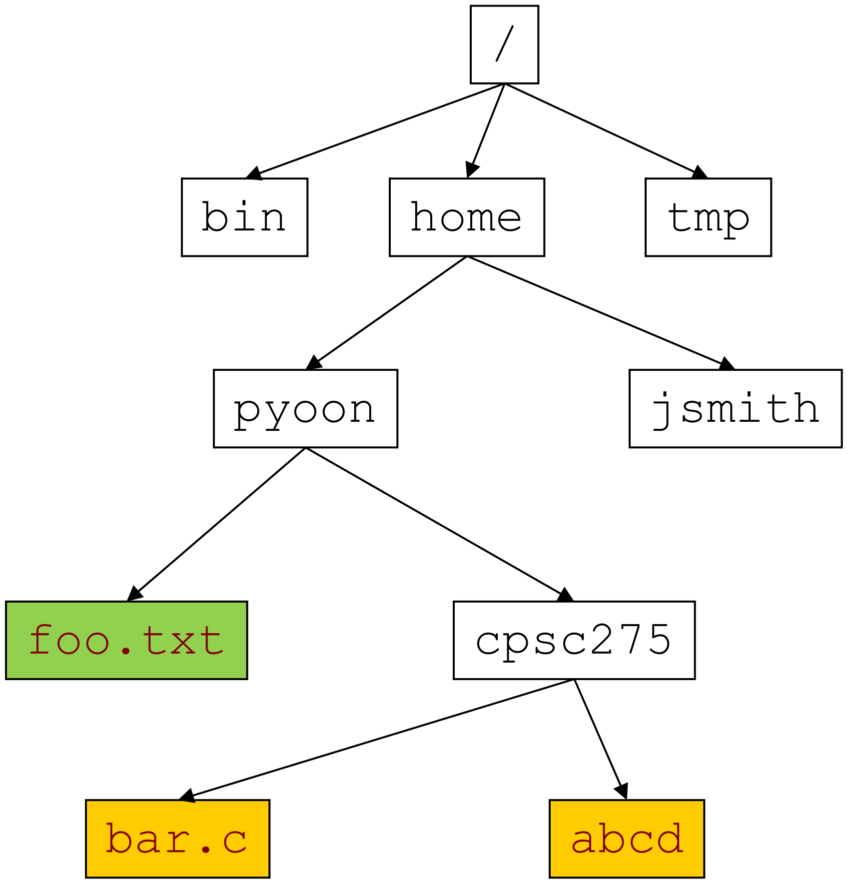
Linux file hierarchy

- Directories may contain plain files or other directories
- Leads to a tree structure for the filesystem
- Root directory: /



Path names

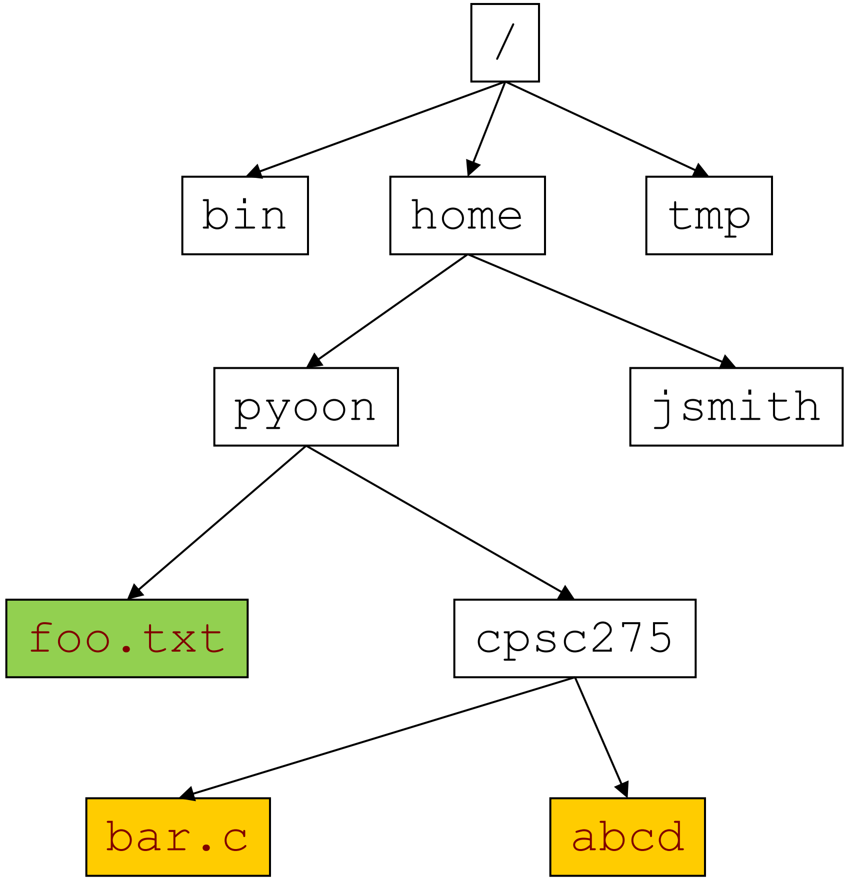
- Separate directories by `/`
- *Absolute path*
  - start at root and follow the tree
  - e.g. `/home/pyoon/foo.txt`



Path names, cont'd

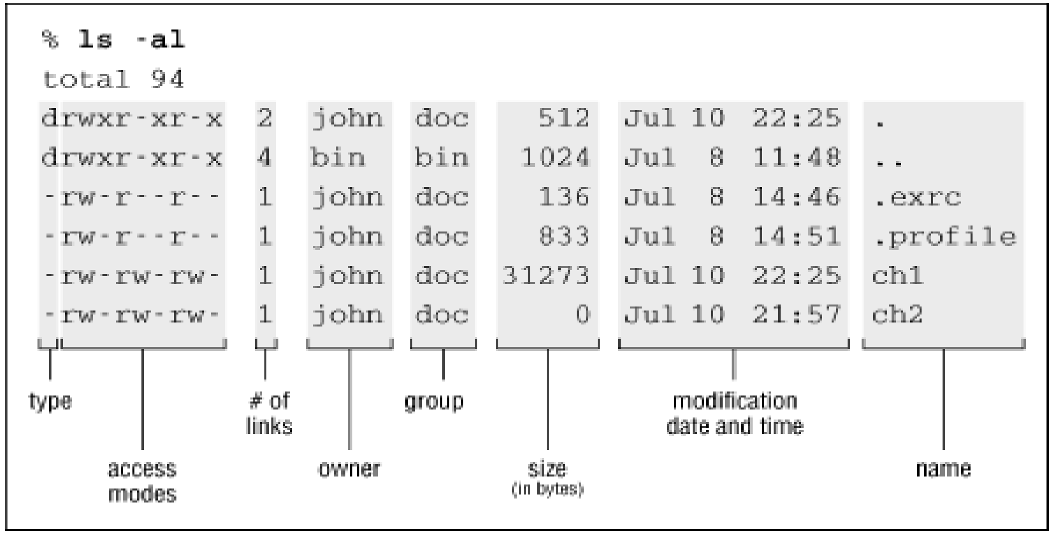
- *Relative path*
  - start at working directory
  - `..` refers to level above;
  - `.` refers to working dir.
  - If `/home/pyoon/cpsc275` is working dir, all these refer to the same file

```
../foo.txt
~/foo.txt
~pyoon/foo.txt
```



Types of files

- Plain (− in the first bit)
  - Most files
  - Includes binary and text files
- Directory (`d`)
  - A directory is actually a file
  - Points to another set of files
- many others …



File permissions

- Permissions used to allow/disallow access to file/directory contents
  - Read (`r`), write (`w`), and execute (`x`)
  - For owner, group, and world (everyone)

- `chmod` <mode> <file(s)>

  `chmod 700 file.txt` (only owner can read, write, and execute)

  `chmod g+rw file.txt`

Basic file I/O

- Processes keep a list of open files
- Files can be opened for reading, writing
- Each file is referenced by a *file descriptor* (integer)
- Three files are opened automatically

  0: Standard Input (`STDIN_FILENO`)

  1: Standard Output (`STDOUT_FILENO`)

  2: Standard Error (`STDERR_FILENO`)

File I/O System Call: `read()`

```
nbytes = read(fd, buffer, count)
```

- reads up to count bytes from file and place into buffer
- `fd`: file descriptor
- `buffer`: pointer to array
- `count`: number of bytes to read
- returns number of bytes read or -1 if error

File I/O System Call: `write()`

```
nbytes = write(fd, buffer, count)
```

- writes count bytes from buffer to a file
- `fd`: file descriptor
- `buffer`: pointer to array
- `count`: number of bytes to write
- returns number of bytes written or -1 if error

Example: A Simple Copy Program, Version 1

```
#define BUFSIZE 1

void main(void)
{
  char buf[BUFSIZE];

  int n;
  while ((n = read(0, buf, BUFSIZE)) > 0)
     write(1, buf, n);
}
```

File I/O system call: `open()`

```
fd = open(path, flags, mode)
```

- `path`: string, absolute or relative path
- `flags`:
  - `O_RDONLY` - open for reading
  - `O_WRONLY` - open for writing
  - `O_RDWR` - open for reading and writing
  - `O_CREAT` - create the file if it doesn't exist
  - `O_TRUNC` - truncate the file if it exists
  - `O_APPEND` - only write at the end of the file
- `mode`: specify access permissions if using `O_CREAT`

Access Permission Bits

`S_IRUSR`	User (owner) can read
`S_IWUSR`	User (owner) can write
`S_IXUSR`	User (owner) can execute
`S_IRGRP`	Group can read
`S_IWGRP`	Group can write
`S_IXGRP`	Group can execute
`S_IROTH`	Others can read
`S_IWOTH`	Others can write
`S_IXOTH`	Others can execute

File I/O system call: `close()`

```
retval = close(fd)
```

- closes an open file descriptor `fd`
- returns 0 on success, -1 on error

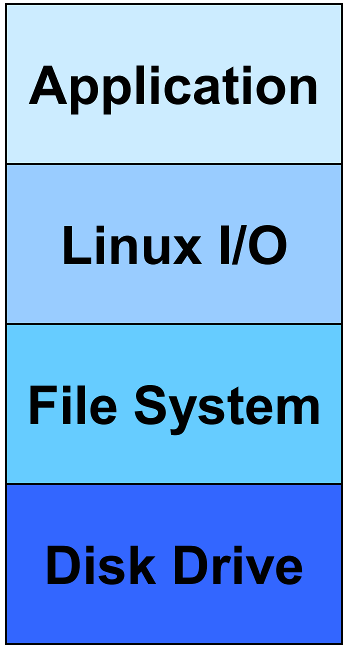
Copy Program, Version 2

```
#define BUFSIZE 1
void main(void)
{
   char buf[BUFSIZE];
   int fdr, fdw, n;

   if ((fdr = open("foo.txt", O_RDONLY, 0)) < 0)
       exit(-1);
   if ((fdw = open("bar.txt", O_WRONLY, 0)) < 0)
       exit(-1);
   while ((n = read(fdr, buf, BUFSIZE)) > 0)
       write(fdw, buf, n);
   close(fdr);
   close(fdw);
}
```

How can we Improve Performance?

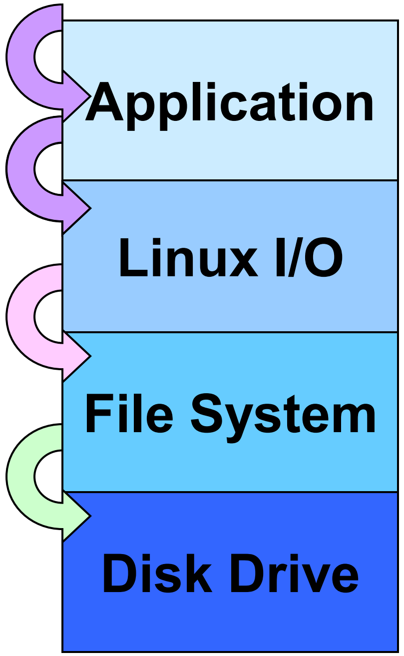
- Making a system call is several orders of magnitude more expensive than a function call
- Given what we know, are there interesting things we can do at the application layer to speed things up?



File System Layering

Caching in the application

- Applications can use caching to improve performance just like the kernel
- Most I/O has both
  - Spatial locality
  - Temporal locality
  - An application level cache in the form of the Standard I/O library attempts to take advantage of this



File System Layering

ANSI C Standard Library

- Collection of high-level input and output functions
- Standard libraries promote cross-platform compatibility.
- Functions, types, and macros of the standard library are accessed by the `#include` directive.
- Actual I/O libraries can be linked *statically* or *dynamically*.

ANSI Standard Libraries

`<assert.h>`	Diagnostics
`<ctype.h>`	Character class tests
`<errno.h>`	Error codes reported by library functions
`<float.h>`	Implementation-defined floating-point limits
`<limits.h>`	Implementation-defined limits
`<locale.h>`	Locale-specific information
`<math.h>`	Mathematical functions
`<setjmp.h>`	Non-local jumps
`<signal.h>`	Signals
`<stdarg.h>`	Variable argument lists
`<stddef.h>`	Definitions of general use
`<stdio.h>`	Input and output
`<stdlib.h>`	Utility functions
`<string.h>`	String functions
`<time.h>`	Time and date functions

I/O data streams

- I/O is not directly supported by C but by a set of standard library functions defined by the ANSI C standard.
  - The `stdio.h` header file contains function declarations for I/O and preprocessor macros related to I/O.
  - `stdio.h` does ***not*** contain the source code for I/O library functions!
- All C character based I/O is performed on streams.
  - All I/O streams must be opened and closed
  - A sequence of characters received from the keyboard is an example of a text stream

File I/O

- General-purpose I/O functions allow us to specify the stream on which they act
- Must declare a pointer to a `FILE` struct for each physical file we want to manipulate

```
FILE* infile;
FILE* outfile;
```

- The I/O stream is "opened" and the `FILE` struct instantiated by the `fopen` function

```
infile = fopen("myinfile", "r");
outfile = fopen("myoutfile", "w");
```

  - Returns pointer to file descriptor or `NULL` if unsuccessful

Standard I/O Streams

- In standard C there are three streams automatically opened upon program execution:

```
FILE *stdin        // input stream
FILE *stdout       // output stream
FILE *stderr       // error messages
```

I/O from Files

- File operation modes are:
  - "`r`" for reading
  - "`w`" for writing (an existing file will lose its contents)
  - "`a`" for appending
  - "`r+`" for reading and writing
  - "`b`" for binary data
- Once a file is opened, it can be read from or written to with functions such as
  - `fgetc` Read character from stream
  - `fputc` Write character to stream
  - `fread` Read block of data from stream
  - `fwrite` Write block of data to stream
  - `fseek` Reposition stream position indicator
  - `fscanf` Read formatted data from stream
  - `fprintf` Write formatted output to stream

`stdio` (caching)

- Each Linux I/O call has a corresponding `stdio` call

```
open()    →  fopen()
close     →  fclose()
read()    →  fread()
write()   →  fwrite()
```

Lab 12

- Implement your own `ls -l` (Linux)
- Both require access to the underlying file system.
- Use Linux system calls
  - `opendir()` – open a directory
  - `closedir()` – close a directory
  - `readdir()` – read directory entries

System Calls vs. Standard Library

- System calls
  - Direct, *unbuffered* requests to the kernel
  - e.g., `write(1,"h",1)`: 1 syscall per character.
  - Precise control
  - Inefficient (Each syscall has overhead)
- Standard Library functions
  - *Buffered* functions
  - Writes to a user-space buffer.
  - Much more efficient (drastically fewer syscalls).
  - Less precise control (e.g., buffering can be tricky). Must `fflush()` to force buffered output to its destination.