CSE 333 – SECTION 4

Quiz1, POSIX I/O Functions, GTK+

Important Dates

- Jan 26th HW2 due
- Feb 3th Midterm
- Feb 6th HW3 due

Basic File Operations

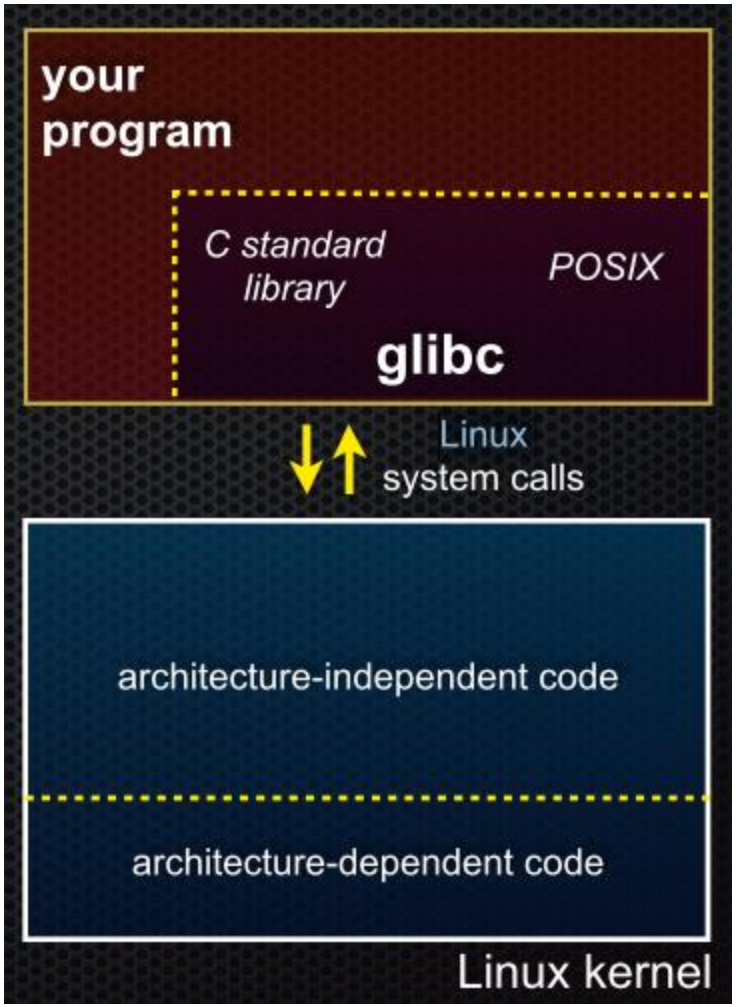
- Open the file
- Read from the file
- Write to the file
- Close the file / free up resources

STDIO vs. POSIX Functions

- User mode vs. Kernel mode.

- STDIO library functions
  - fopen, fread, fwrite, fclose, etc.
    use FILE* pointers.

- POSIX functions
  - open, read, write, close, etc.
    use integer file descriptors.



System I/O Calls

```
int open(char* filename, int flags, mode_t mode);
```

Returns an integer which is the file descriptor.
Returns -1 if there is a failure.

filename: A string representing the name of the file.
flags: An integer code describing the access.

- O_RDONLY -- opens file for read only
- O_WRONLY – opens file for write only
- O_RDWR – opens file for reading and writing
- O_APPEND --- opens the file for appending
- O_CREAT -- creates the file if it does not exist
- O_TRUNC -- overwrite the file if it exists

mode: File protection mode. Ignored if O_CREAT is not specified.

[man 2 open]

System I/O Calls

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

fd: file descriptor.

buf: address of a memory area into which the data is read.

count: the maximum amount of data to read from the stream.

The return value is the actual amount of data read from the file.

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

Returns 0 on success, -1 on failure.

```
[man 2 read]
[man 2 write]
[man 2 close]
```

Errors

- When an error occurs, the error number is stored in **`errno`**, which is defined under `<errno.h>`
- View/Print details of the error using **`perror()`** and **`errno`**.
- POSIX functions have a variety of error codes to represent different errors. Some common error conditions:
  - **`EBADF`** - *fd* is not a valid file descriptor or is not open for reading.
  - **`EFAULT`** - *buf* is outside your accessible address space.
  - **`EINTR`** - The call was interrupted by a signal before any data was read.
  - **`EISDIR`** - *fd* refers to a directory.
- errno is shared by all library functions and overwritten frequently, so you must read it right after an error to be sure of getting the right code

```
[man 3 errno]
[man 3 perror]
```

Again, why are we learning POSIX functions?

- They are unbuffered. You can implement different buffering/caching strategies on top of read/write.
- More explicit control since read and write functions are system calls and you can directly access system resources.
- There is no standard higher level API for network and other I/O devices.

Read the man pages

- **man, section 2: Linux system calls**
- `man 2 intro`
- `man 2 syscalls`
- `man 2 open`
- `man 2 read`
- …
- **man, section 3: glibc / libc library functions**
- `man 3 intro`
- `man 3 fopen`
- `man 3 fread`
- `man 3 stdio` for a full list of functions declared in `<stdio.h>`
- …

Read the man pages

- Be sure you're reading the correct man page for a specific call.

- Ex. If you write "man read" you'll get the shell command rather than the system call

- `[Man man]` You can see the system calls are in section 2
- `[Man 2 read]` Here's the system call read.

Reading a file

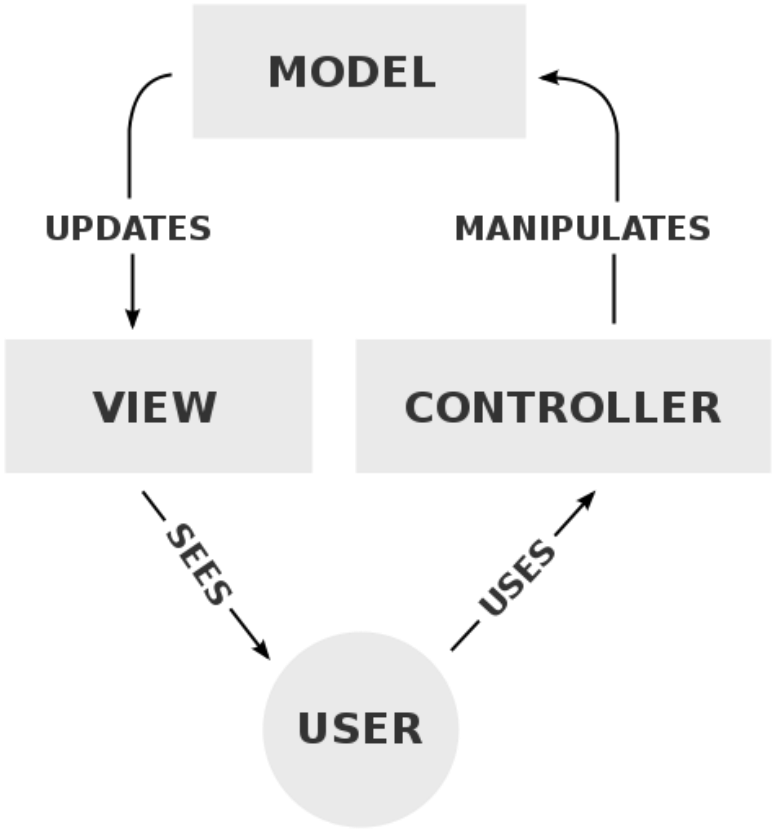
```
#include <errno.h>
#include <unistd.h>

...

  char *buf = ...;
  int bytes_left = n;
  int result = 0;

  while (bytes_left > 0) {
     result = read(fd, buf + (n-bytes_left), bytes_left);
     if (result == -1) {
       if (errno != EINTR) {
        // a real error happened, return an error result
       }
       // EINTR happened, do nothing and loop back around
       continue;
     }
     bytes_left -= result;
  }
```

HW3 : MVC, GTK+

- HW3 online now.
- You must work in groups.

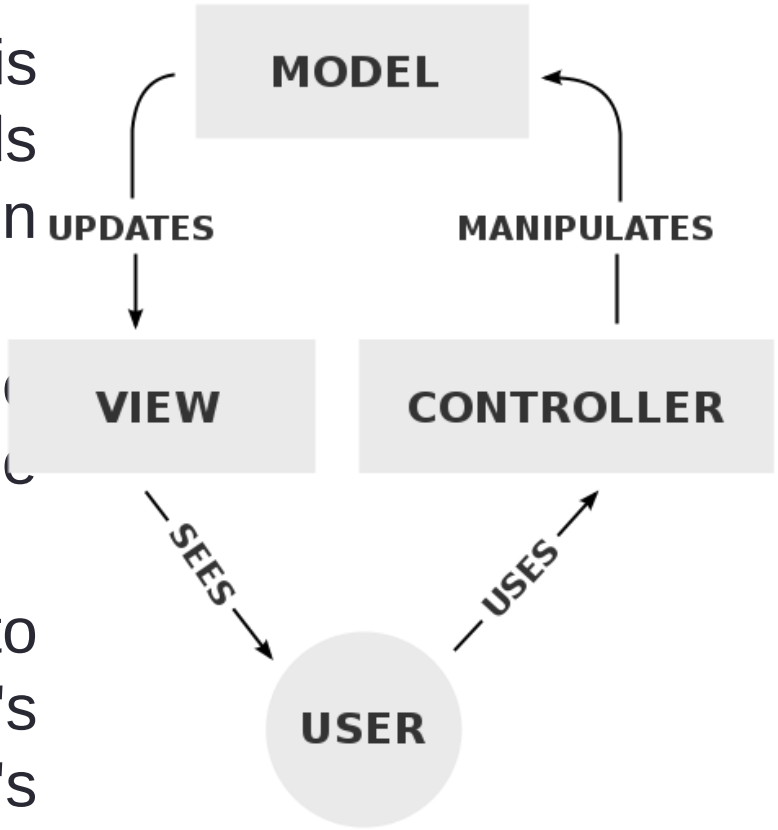
Model–view–controller (MVC)

- The *model* directly manages the data, logic, and rules of the application.
- A *view* can be any output representation of information, such as a chart or a diagram.
- The *controller* accepts input and converts it to commands for the model or view.



MVC interactions

- A *model* stores data that is retrieved according to commands from the controller and displayed in the view.
- A *view* generates new output to the user based on changes in the model.
- A *controller* can send commands to the model to update the model's state. It can also change the view's presentation of the model



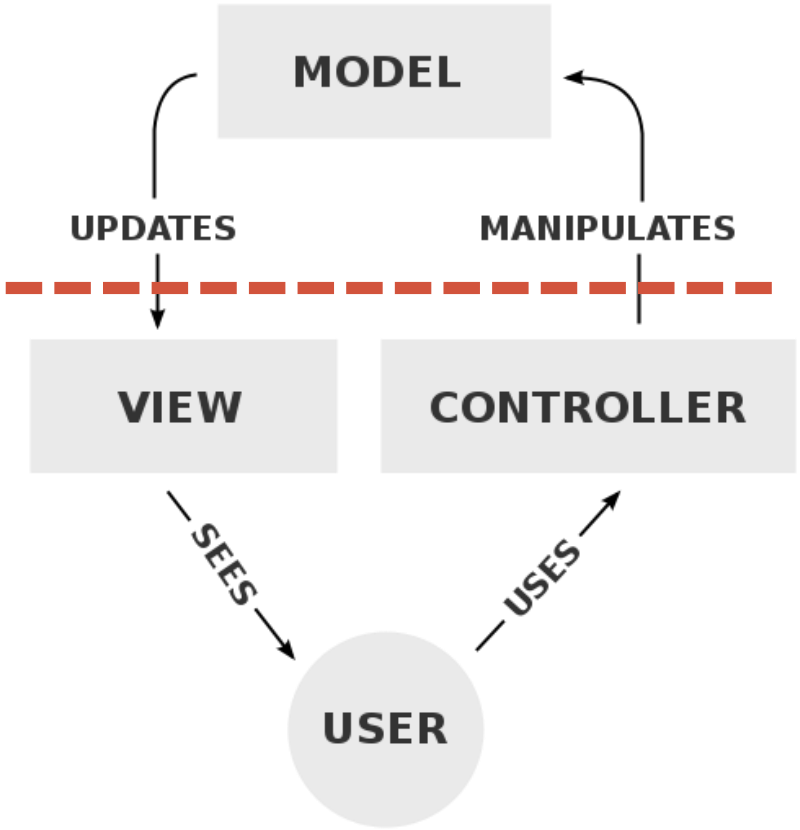
See HW3

- https://courses.cs.washington.edu/courses/cse333/16au/assignments/hw3/hw3.html

- Make sure you can display a board, can provide some way for the user to select and swap adjacent candies, and can update the number of moves left field.

MVC version of Candy Crush

- Eventually you will be splitting the view/controller from the model, across the Internet.



GTK+

- **GTK+** is Installed on attu

- The **X Window System** (**X11**, or **X**)
  - A windowing system for bitmap displays, common on UNIX-like computer operating systems
  - Provides the basic framework for a GUI environment

- For you to remotely use GTK+ and run X11 applications on MAC/Linux
  - 1) SSH –X usr@attu.cs.washington.edu
  - 2) X-Server
    - X11

- For you to remotely use GTK+ and run X11 applications on Windows, we need 2 additional pieces of software.
  - 1) SSH Client
    - Eg. PuTTY
  - 2) X-Server
    - Eg. Xming X-Server
    - Another option: Cygwin/X

Xming

- Download: https://sourceforge.net/projects/xming/

- 1. Double click on the **Xming** shortcut on the desktop 

Note: If you have a firewall installed on your computer you will need to allow remote hosts access to the X-server

- 2. After a short while, you will see the **X** logo in the system tray.
- 3. Launch **Putty** and check '*Enable X11 forwarding*' under *SSH*.

GTK+ features

- Basic drawing model
- Hierarchical containers
- Reference counted (but mostly you don't see it)
- Event driven

GTK+ intro example

- Getting started
- https://developer.gnome.org/gtk3/stable/gtk-getting-started.html#id-1.2.3.5

Event driven

- While the program is running, GTK+ is receiving *events*.
  - Typically input *events* caused by the user interacting with your program
  - Could also be messages from the window manager or other applications

Signals may be emitted on your widgets

Connecting handlers for these signals => respond to user input

Hierarchical containers

- Example:
- https://developer.gnome.org/gtk3/stable/GtkGrid.html

Object Hierarchy

```
GObject
└── GInitiallyUnowned
    └── GtkWidget
        └── GtkContainer
            └── GtkGrid
```

Reference counted

- https://developer.gnome.org/gobject/stable/gobject-memory.html

- g_object_unref ()