



Operating Systems

Section 1 - C, GDB, Course Intro
9/29/22

Overview

- 1) Info about labs
- 2) Brief recap of 351/333 topics
- 3) Tools for debugging
- 4) TA procedure: Office hours, discussion board
- 5) Misc. info

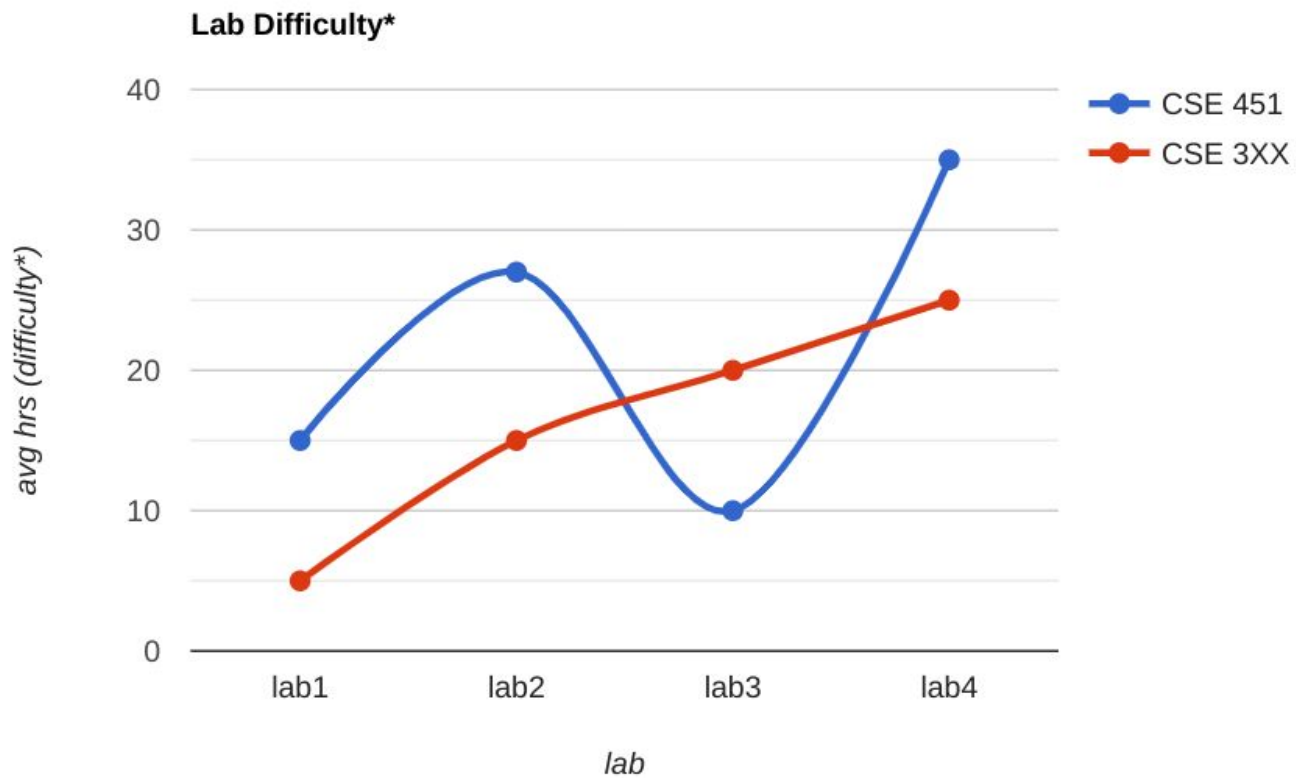




Labs

There are 4 labs:

1. File System Calls (Coming out next Monday!)
2. Processes and Pipes
3. Memory
4. File System





Why should you start Lab 1 early?

- It takes time to get used to qemu and xk
- Create your own file info struct
 - Have to figure out what fields are needed
- Compile Time Issues
- Getting comfortable with gdb

Time to complete varies between 5 hrs and 20 hrs

Part 1: The C Programming Language



What Was C, Again? A Brief Recap

To jog your memory, not to re-teach C. Skimming over 351/333 isn't a bad idea

- Functions & Structs (they exist, and are about as complex as C gets)
- Pointers & Memory (to * or not to *, that is a question)
- Forward Declarations & Header files (working with multi-file projects)
- The Preprocessor (and how it relates to header files)
- Assembly



Functions (code to call), Structs (bundle of state)

```
// function, like in most programming languages
int sum3(int x, int y, int z) {
    return x + y + z;
}

// not a class: only public fields, no inheritance or methods
// typedef lets you refer the struct as "struct Point2D", or just "Point2D"
typedef struct Point2D {
    double x;
    double y;
} Point2D; // These names happen to match, but they don't have to

double dot(struct Point2D point1, Point2D* point2) {
    return point1.x * point2->x + point1.y * point2->y;
}
```




Pointers & Addresses

- **&:** Gets the address of where something is stored in (virtual) memory
 - a 32/64 bit (4/8 byte) number
 - you can do arbitrary math to a pointer value (might end up with an invalid address.....)
- *****: Dereferencing, “give me whatever is stored in memory at *this* address”.
 - dereferencing invalid addresses (nullptr, random address) causes a segfault!

**** A decent chunk of bugs are basically passing pointers when you shouldn't and vice versa****



Pointers & Addresses

```
void increment(int* ptr) {  
    *ptr = *ptr + 1;  
}
```

```
int x = 3;  
increment(&x);  
// x is now 4
```

← Pass in a pointer: the address at which some int is stored
*ptr gets the value stored at the address stored by ptr
So we assign to the memory at ptr's address:

“whatever was there before + 1”

The pointer (address) is passed by value: “*ptr = *ptr + 1;
” only changes the local “ptr” variable

← Use the address at which ‘x’ resides in memory



Pointers & Addresses

```
void class_string(char** strptr) {  
    *strptr = "class";  
}
```

```
char str[6] = "hello"; // why 6?
```

```
char* str2 = str;
```

```
class_string(&str2); // what would printf(str2) output?
```



Pointers & Addresses

```
void random_coordinate(int* x, int* y, int* z) {  
    *x = rand() % 100;  
    *y = rand() % 100;  
    *z = rand() % 100;  
}
```

```
int x, y, z;  
random_coordinate(&x, &y, &z);
```



Function Ordering

- C compiler is single pass
 - If you define function A, then function B, the compiler doesn't know about B until it's done reading A
- This will have a compiler error: when reading `get4()`'s implementation, `get3()` is unknown

```
int get4() { return get3() + 1; }
```

```
int get3() { return 3; }
```



Solution: Forward Declarations, Header Files

- The solution? Declare things before defining them

```
int get3(); // There will be a function get3 with this signature
int get4(); // Also one called get4()
```

```
int get4() { return get3() + 1; } // Now this is okay: we
promised the compiler that get3() will exist
int get3() { return 3; }
```

- We end up putting our forward declarations in a **header file** so that we know everything is declared first. As a bonus, other code can reference the header file to use functions it declares



Forward Declarations of Global Variable

```
/* === header.h === */
extern int var; // declare a variable without allocation

/* === program.c === */
#include "header.h"
int var; // define (allocate) a variable

int get() {return var;}

/* === another_program.c === */
#include "header.h"

// Don't define the variable again! Variable allocated in "program.c"
int get2() {return var * 2;}
```



Header Files & The Preprocessor

Now we have two problems:

1. Implementations don't have the forward declarations anymore (we moved to a new file)
 - a. Solution: The Preprocessor `#include "MyHeader.h"` in effect, replace this line with the entire content of MyHeader.h
2. Duplicated declarations: if the header file is included in multiple places, we can end up declaring the same function signature multiple times (since `#include` is copy-paste)
 - a. Solution: Header Guards, everything between the `ifndef` and `endif` is only expanded once

```
// mymath.h
#ifndef MYMATH
#define MYMATH
int get4();
int get3();
#endif
```




Preprocessor Macros to Know

`#include`: embed the given file *here*. As in, copy-paste the whole thing.

`#define A` (or `#define A B`): register A as a known symbol. If B is given, replace all occurrences of A with B
-> Used for constants! (e.g. “`#define SIZE 20`”)
-> Also used for macros. e.g. “`#define MAX(a,b) (a) > (b) ? (a) : (b)`”
This is a *find/replace* operation. Be careful of the operator precedence!

`#if ___ / #endif`: Only include the code between the `#if` and `#endif` if the condition is true

`#ifndef ___ / #endif ___ / #endif`: Only include the code between this and `endif` if the symbol is/isn't defined

Part 2: Tools For Debugging



Old Friend: Printf

Prints are very useful for simple debugging:

- How far have we reached in a function?
- How many times did we meet a condition?
- Function invocations & its parameters

However, sometimes prints are not enough:

- bugs in your code can impact printf's in unexpected ways
- printf grabs a console lock that may make the bug difficult to reproduce
- printf uses a buffer internally, so prints might be interleaved
- can't print in assembly



New Friend:

GDB

**This is a systems class and you'll be doing a LOT of debugging
Also lots of pointers.
Really, the pointers are the main reason for the debugging**



GDB commands to know: a non-exhaustive list

`gdb path/to/exe`

`run`: start execution of the given executable

`n`: run the next line of code. If it's a function, execute it entirely.

`s`: run the next line of code. If it's a function, *step* into it

`c`: run the rest of the program until it hits a breakpoint or exits

`b _____`: set a breakpoint for the given function or line (e.g. “`b myfile.c:foo`” or “`b otherfile.c:43`”)

`bt`: get the stack trace to the current point. Can be ran after segfaults!

`up/down`: go up/down function stack frames in the backtrace

`(r)watch _____`: set a breakpoint for the given thing being accessed

`p _____`: print the value of the given thing

`x _____`: examine the memory at an address. Many flags

GDB Example

```
1 #include <stdio.h>
2
3 void increment(int *ptr) {
4     if (ptr == NULL) {
5         exit(1);
6     }
7     *ptr += 1;
8 }
9
10 int main() {
11     int a, b, c;
12
13     printf("starting value for a: %d, b: %d, c: %d\n", a, b, c);
14     increment(a);
15     increment(a);
16
17     increment(NULL);
18     return 0; // never reaches here
19 }
20
```

```
Reading symbols from a.out...done.
(gdb) b main
Breakpoint 1 at 0x40060d: file example.c, line 13.
(gdb) b 5
Breakpoint 2 at 0x4005e9: file example.c, line 5.
(gdb) run
Starting program: /homes/iws/jlli/a.out

Breakpoint 1, main () at example.c:13
13     printf("starting value for a: %d, b: %d, c: %d\n", a, b, c);
(gdb) print a
$1 = 0
(gdb) print b
$2 = 0
(gdb) print c
$3 = 32767
(gdb) n
starting value for a: 0, b: 0, c: 32767
14     increment(a);
(gdb) c
Continuing.

Breakpoint 2, increment (ptr=0x0) at example.c:5
5     exit(1);
(gdb) bt
#0  increment (ptr=0x0) at example.c:5
#1  0x0000000000400634 in main () at example.c:14
(gdb)
```



General Debugging Tips

- Get familiar with GDB
 - Stepping through line by line and printing out variables is slow, **but will find the bug.**
- Make sure you know what the code is supposed to do first
 - There are a lot of complicated systems, with limited framework. Unlike 333, this isn't fill-in-the-blank
- Should still use printf's
 - It can be an efficient way to find what section of code is wrong so your GDB debugging can be more focused
- GDB step by step tutorials online
- GDB [cheat sheet](#)

You will get a chance to practice with GDB in Lab 1 :)

Part 3: Procedure



Regarding office hours

- There are a *lot* of strange ways you can break xk
- Unlike in other classes, there are many functional ways to structure your code (no one right answer)
- Going through GDB in office hours is way too slow

- Please do preliminary debugging as far as you can before office hours, so we can give useful advice
- For particularly weird issues, we might not be able to solve your bug within available time constraints



Discussion Board

If you've tried debugging and have come up against a wall that would take too long for office hours, consider posting on the discussion board.

Include **DETAILS**

- What is the problem
- Which methods does it manifest in
- What does work
- **What debugging have you tried, & what did you find**

Our time is limited and there are a lot more students than TAs, so our ability to be helpful is directly influenced by the quantity of useful debugging information you provide.



Reminders

- Find a lab partner and fill out the form by next Monday
- Lab comes out next Monday
- Readings due every class



Questions (C, GDB, the class, your TA, etc)