Operating Systems

Section 1 - Course Introduction, Misc. Topics
31 March 2022

Lab 1 is out!

- Partner sign-up form on Ed
  - Only one partner per group needs to fill it out\
  - Fill out ASAP if you want TA-assigned partner
  - Check your email for incoming messages
- Due Friday 1/15 (in two weeks)

Schedule

1) Brief Lab 1 Intro
2) Brief recap of 351/333 topics
3) TA procedure: Office hours, discussion board
4) Misc. info

Where to start?

https://gitlab.cs.washington.edu/xk-public/22sp/blob/main/lab/lab1.md
Start by reading:

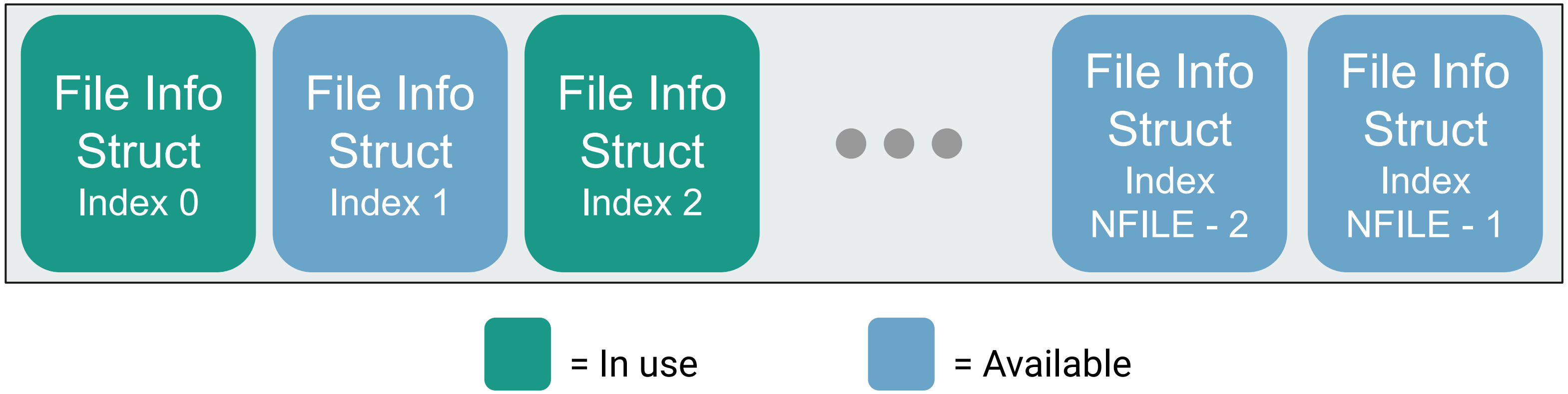
- **lab/overview.md** - A description of the xk codebase. A MUST-READ!
- **lab/lab1.md** - Assignment write-up
- **lab/memory.md** - An overview of memory management in xk
- **lab1design.md** - A design doc for the lab 1 code
  - You will be in charge of writing design docs for the future labs (which will be a bit more comprehensive than the one provided for lab 1). Check out lab/designdoc.md for details.

File Information

Need a way to store the following information about a file:

- In memory reference count
- A pointer to the inode of the file
- Current offset
- Access permissions (readable or writable) [for when we add pipes and file writability later]

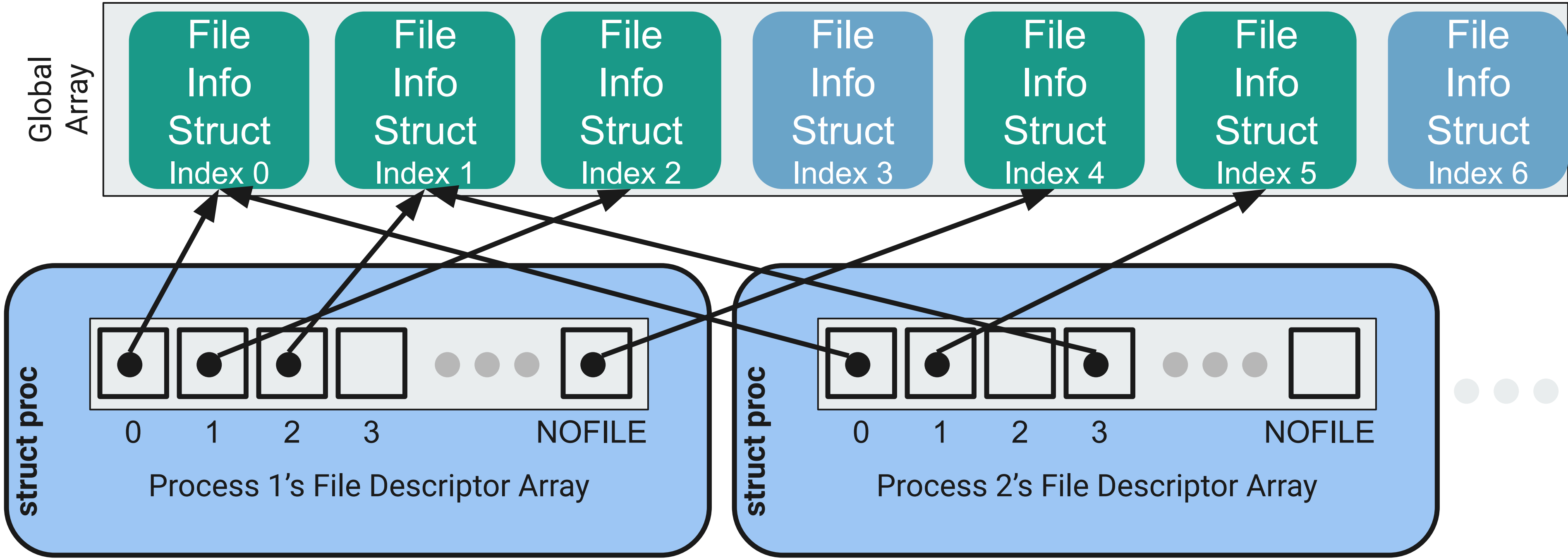
Kernel View



There will be a global array of all the open files on the system (bounded by NFILE) placed in static memory.

fd = ***index*** into local File Descriptor Array

Process View



Functions

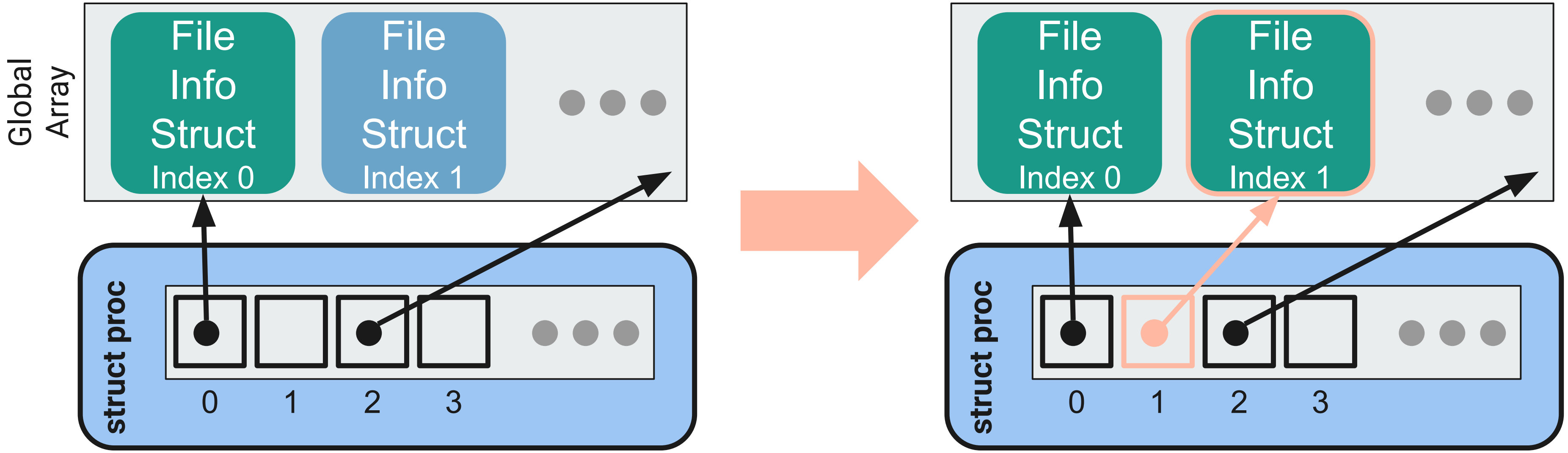
filewrite and *fileread*

- Writing or reading of a "file", based on whether the file is an inode or a pipe.
  - Note that file is in quotes. A file descriptor can represent many different things. You could be reading from a file, or you could be reading from standard in or a pipe!
- Don't need to worry about the pipe part for this lab, just the inode files.
- Hint: check out the functions *readi* and *writei* defined in kernel/fs.c
  - file layer provides "policy" for accessing files, inode layer provides "mechanism" for reading/writing

fileopen

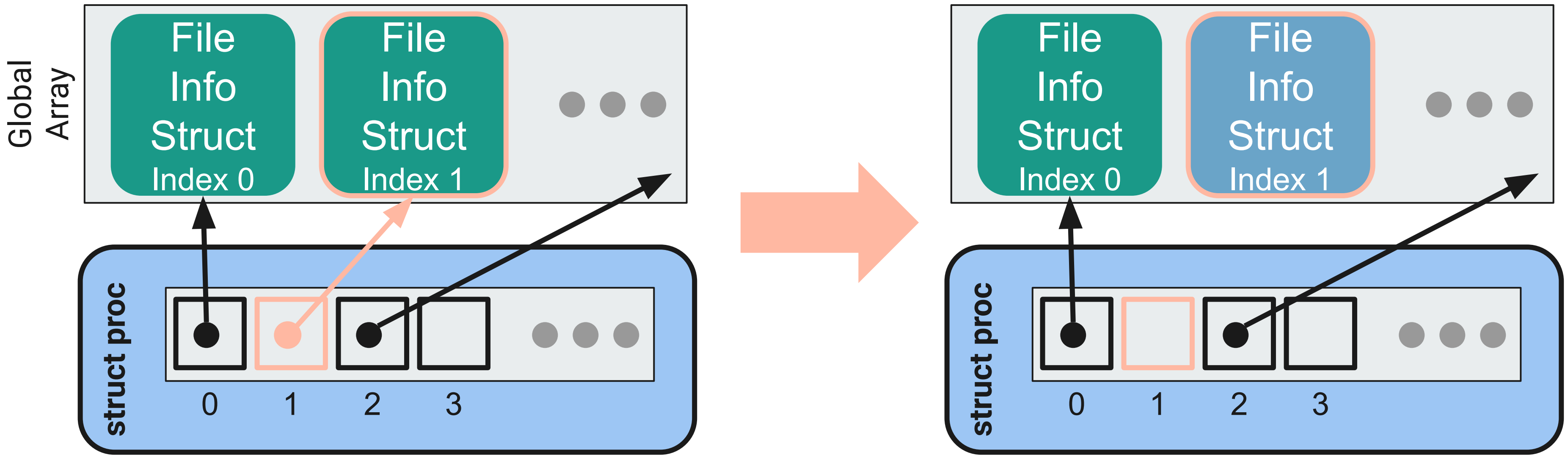
Finds an open file in the global file table to give to the process

Hint: take a look at namei()



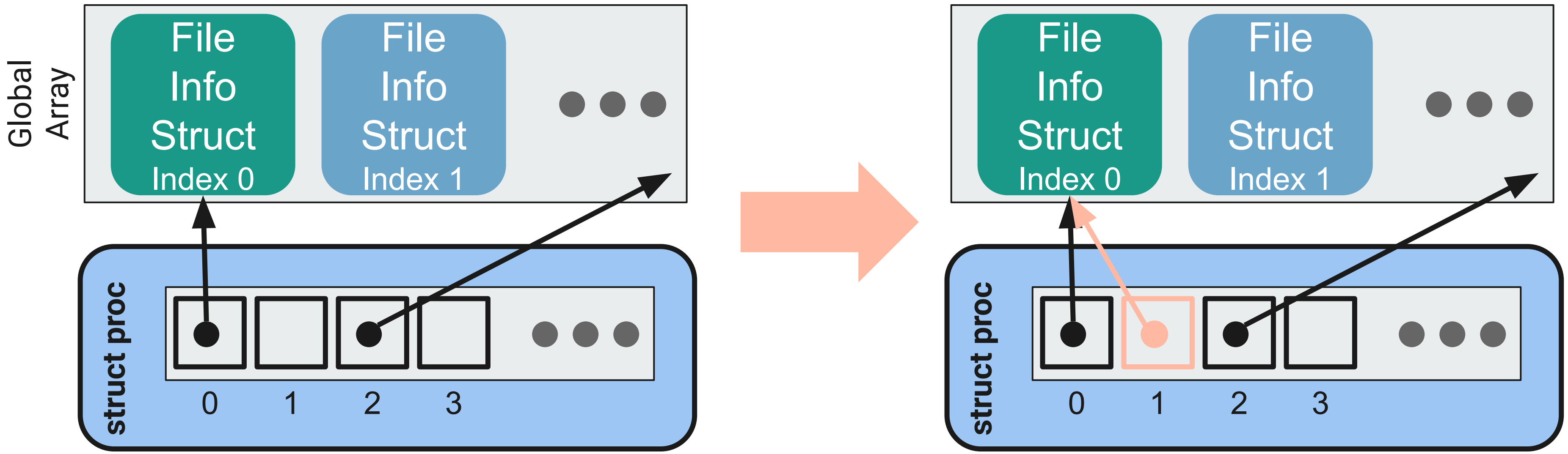
fileclose

Release the file from this process, will have to clean up if this is the last reference



filedup

Duplicates the file descriptor in the process' file descriptor table



Part 2: 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)
- Forward Declarations & Header files (working with multi-file projects)
- The Preprocessor (and how it relates to header files)
- Pointers & Memory
- Assembly

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

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

typedef struct s_Point2D { // Not a class: only public fields. No
inheritance, no methods
 double x;
 double y;
} Point2D ; // Because of typedef, can use either type "struct
s_Point2D", or type "Point2D"

double dot(struct s_Point2D point1, Point2D point2) { // Pass by value
 return point1.x * point2.x + point1.y * point2.y;
}
```

Forward Declarations

- 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

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

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

Forward Declarations, Header Files

- The solution? Declare things before defining them

```
int get3(); // There will exist a function called get3 with no args, returning int
int get4(); // Also one called get4()

int get4() {  return get3() + 1; } // This is okay: we know 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

```c
/* === 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)

    Solution: The Preprocessor: **#include “MyHeader.h”**

    - In effect, replace this line with the entire contents of MyHeader.h
    - Now, implementation can reference things the header declares

  2) Duplicate declarations: if we include the same header multiple times (in order to references its contents from multiple places), have repeated code -- since #include is copy-paste

    Solution: Header Guards (#ifndef AAA \n #define AAA \n …… #endif)

      Everything between the ifndef and endif is only duplicated once

Header Files & The Preprocessor (2)

```
// mymath.h
#ifndef MYMATH // Header guard: in effect
#define MYMATH // only include declaration
int get4();            // once
int get3();
#endif
```

```
//mymath.c
#include "mymath.h" // declare get3/4
int get4() { return get3() + 1; }
int get3() { return 3; } // Compiler binds
                 // implementation to declaration
```

```
// morecode.c
#include <stdio.h>
#include "mymath.h" // Have get3's declaration
void print3() { printf("%d", get3()); } // prints 3
```

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

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

Pointers & Addresses

Get the address of where something is stored in (virtual) computer memory

-> a 32/64 bit (4/8 byte) number.

-> Just a 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”.

If you use an address outside of what your program has claim to, segfault.

Pointers & Addresses (2)

```
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

Memory

Program memory:
- The Stack
  - Function data
  - Tied to function lifetime
  - Grows down
- The Heap
  - Anything dynamically allocated
  - Persists until deleted
  - Grows up… ish

System memory:
- Registers
  - The CPU has a few places to store actively-used data
- RAM
  - Volatile data associated with a running program
- Cache
  - CPU hs caches to make lookups faster

All CPU operations must be done on data in registers

Compilation & Assembly

C code is human-readable (mostly). It’s not machine-readable -- you can’t just feed the CPU some C.

Need to **compile** the code from C into machine code (ones and zeros)

-> It’s just a bunch of simple instructions. Add *this* and *that*. Move this data over there.

Assembly is raw CPU instructions in a slightly-readable format.

This class:

- You won’t need to write assembly
- You *will* need to read a bit of assembly (e.g. kernel/trapasm.S)
- You *will* need to use registers, understand how they work, and understand how x86 does function calls

We’ll go over some of the details for relevant assignments, but skimming over CSE351 would not hurt.

Compilation Process

1) **Preprocessor**

    Scan all files top-to-bottom, doing text substitution as appropriate

2) **Compiler**

    Compile C files into assembly files (intermediate form)

3) **Assembler**

    “Assemble” assembly code into machine code (creating object files)

4) **Linker**

    Combine all source files together into executable. Include external libraries, too

Wow, that was a lot of review, very quickly

I assume many of you have questions

Please ask.

Your best friend this quarter

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

General Debugging Tips (more to come later)

- 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
- Don't be afraid of println debugging
    - It can be an efficient way to find what section of code is wrong so your GDB debugging can be more focused
    - Some timing-based bugs don't really show up in GDB. Need printlns to figure out problematic orderings
- Get familiar with GDB
- Get familiar with GDB
- Get familiar with GDB

Debugging Tips

Brennan linked a PDF in the stickied post on discussion board.

It goes over some of the GDB commands you’ll find useful, plus some discussion of common issues people need to try to debug. It’s worth reading now and probably re-reading halfway through lab 2.

Part 3: Procedure

Regarding office hours

- Zoom is not a great platform for helping multiple people at the same time
- There are a *lot* of strange ways you can break xk
- Unlike in other classes, there are many functional ways to structure your code
- Going through GDB in office hours is way too slow

What are we saying?

- This is a 400-level class, and so you're expected to be able to reason through and debug your own code.
- 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
- Which code do you know works
- **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.

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