CSE 451: Section 1

C, GDB, Lab 1 intro
1/4/24
Overview

1) Review of C
2) Tools for debugging
3) Office hours, discussion board
4) Lab 1 intro
Review of C
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……)
    - Ptr++ Increments address by the size of the pointed to type
    - no pointer arithmetic on a void pointer!

- *: 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

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

void example() {
    int x = 3;
    increment(&x); // value of x?
}
```

← Pass in a pointer
ptr = address of an int
*ptr = value stored at the address ptr

← Gets the address at which 'x' resides in memory
Pointers & Addresses

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

void example() {
    char* str = "hello"; // what would strlen(str) return?
    char* str2 = str;
    class_string(&str2);  // what would printf(str2) output?
}
```
Find the bug

```c
struct elem {
    int value;
    struct elem *next;
};

int example(struct elem* e) {
    if (e != NULL) {
        return e->next->value;
    }
    return -1;
}
```
Find the bug 🐛

```c
struct elem {
    int value;
    struct elem *next;
};

void increment(struct elem *e) {
    if (e != NULL) {
        e->value += 1;
    }
}

void example() {
    struct elem *e;
    increment(e);
}
```
Find the bug

```c
struct elem {
    int value;
    struct elem *next;
};

struct elem* alloc_elem() {
    struct elem e;
    return &e;
}

void example() {
    struct elem* e = alloc_elem();
    if (e != NULL) {
        e->value = 0;
    }
    // ...
}
```
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 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.
  - `ni`: Same behavior, but goes one assembly instruction at a time instead.
- `s`: run the next line of code. If it’s a function, `step` into it
  - `si`: Same as “s”, but goes one assembly instruction at a time instead.
- `c`: run the rest of the program until it hits a breakpoint or exits
GDB commands to know: a non-exhaustive list

- `b _____`: set a breakpoint for the given function or line (e.g. “b file.c:foo”)
- `bt`: get the stack trace to the current point
- `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
  - Can understand C-style variable syntax, e.g.: `p *((struct my_struct*) ptr)` interprets the memory pointed to by ptr as a `struct my_struct`.
- `x _____`: examine the memory at an address. Many flags
GDB Example

```
#include <stdio.h>

void increment(int *ptr) {
  if (ptr == NULL) {
    exit(1);
  }
  *ptr += 1;
}

int main() {
  int a, b, c;

  printf("starting value for a: %d, b: %d, c: %d\n", a, b, c);
  increment(a);
  increment(a);

  increment(NULL);
  return 0; // never reaches here
}
```
GDB Cheatsheet

See this GDB cheatsheet for a good overview of what’s possible:
Logistics
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 (What did you expect to see? What actually happened?)
- 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 tomorrow!
- Read through lab 1 handout and other relevant docs
Lab 1 Intro
What is xk?

- xk stands for “experimental kernel”
- Configured to run on qemu (hw emulator)
- A simpler version of the early linux kernel
- 64 bit port of xv6
Different components of the xk kernel *(roughly)*

- **Syscalls**
- **File System**
  - file.c deals with open files management and managing the file info struct (lab1)
  - fs.c deals with writing and reading blocks from disk and other helper functions (lab4)
- **Processes**
  - fork/exec/wait implementation
  - proc.c and exec.c (lab 2)
- **Memory management**
  - writing the page fault handler (for stack, heap, and else) , trap.c (lab3)
Where to start?

https://gitlab.cs.washington.edu/xk-public/24wi

Some suggested reading:

- **lab/lab1.md** - Assignment write-up (definitely read this one)
- **docs/xk/overview.md** - A description of the xk codebase (reference; skimming the baseline code walkthrough section might be helpful)
- **docs/xk/memory.md** - An overview of memory management in xk (mostly relevant for lab 3)
- **docs/xk/debugging.md** - A guide to understanding error messages
- **lab/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.
Summary of Lab 1

- Setup your xk repo
- Read and learn about existing code
- Support file API (through syscalls)
  - syscall validation (checking for valid args etc.)
  - open file (I/O) abstraction
    - user: file descriptor
    - kernel: file_info
File API

fd = open(filename)
Returns a per-process handle to be used in subsequent calls (implemented as a C int)
Shell pre-assigns stdin, stdout as file descriptors (0, 1)

read/write(fd, buffer, numBytes)
Read or write numBytes into/out of buffer, changes position in file

new_fd = dup(fd)
Make a new file descriptor, copy of the previous one (used in shell)

close(fd)
We’re done with using this file descriptor
File API

● **File descriptors**
  ○ used for all I/O, eg, network sockets, pipes for interprocess communication
  ○ applications use read/write regardless of what it is reading/writing to
  ○ per-process
    ■ but can be passed between processes
    ■ inherited by child processes
    ● important for how fork/exec and the shell works
    ● examples: \texttt{ls | wc} \texttt{ls > tmpfile} \texttt{wc < tmpfile}

● **Kernel *should not* trust file descriptor (might not be previously opened, etc.)**
  ○ applications should not be able to crash kernel
You will need to implement a number of file related system calls.

Implementing syscalls consists of two steps:

- parsing and validating syscall arguments
  - see implemented syscalls for reference (sysfile.c)
  - argptr, argstr, argint, what do these functions do?
- perform the requested file operations
  - need to write your own file operations using the provide inode layer (file.c)
File Descriptors - Kernel View

- Kernel needs to give out file descriptors upon open
  - must be give out the smallest available fd
  - fds are unique per process
    - e.g. fd 4 in process A can refer to a different file than fd 4 in process B
  - there's a max number (NOFILE) of open files for each process
    - each process should know its fd to file mapping

- Kernel needs to deallocate file descriptors upon close
  - close(1) means that fd 1 is now available to be recycled and given out via open
The current xk file system only implements a primitive inode layer, so you need to create a file abstraction yourself. You need to track at least the following information for each open file:

- In memory reference count
- A pointer to the inode of the file
- Current offset
- Access permissions (readable or writable)
Allocation of File Structs

After defining the file struct, you need a way to allocate it.

You can statically allocate an array of file structs.
Inode Layer

iopen() = looks up an inode using a given path (populates and loads inode into memory if necessary), increments the inode’s reference count

irelease() = decrements this inode’s reference count (internally, once the reference count is 0, this inode is removed from the inode cache)

readi() / concurrentreadi() = read data using this inode

writei() / concurrentwritei() = write data using this inode

locki() and unlocki() = locks or unlocks the inode (this does NOT change the inode’s reference count)

File layer provides “policy” for accessing files, inode layer provides “mechanism” for reading/writing

Note: For Lab 1, it is likely not necessary to call locki() or unlocki() directly
Lab 1: Start Early!

- It takes time to set up and navigate the code base
- Compile Time Issues
- Getting comfortable with gdb
Git Resources

- Git manual: https://git-scm.com/docs/user-manual
- Git tutorial: https://learngitbranching.js.org/?locale=en_US