CSE 451: Section 1

C, GDB, Lab 1 intro
3/28/24

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 64 bit (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

```
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

```
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 🐛

```
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:

- printfs may affect bugs in your code 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

- 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 till 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
    - understands 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

```c
#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
  increment(a);
  increment(a);

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

```
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)
```

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 introduce bugs in the kernel
- Please do preliminary debugging as far as you can before office hours, so we can give useful advice
    - should know what test case in what scenario is failing
    - if a function returns a different value than expected, figure out what line caused the issue (is a strcmp failing? is a NULL ptr check failing?)
- We may ask you to find out some information about your error before getting back to you

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

Reminders

- Find a lab partner and fill out the form by today!
- Read through lab 1 handout and other relevant docs

Lab 1 Intro

What is xk?

- xk stands for “e**x**perimental **k**ernel”
  - the teaching OS you will be extending throughout the quarter
  - needs to understand different parts of the codebase for each lab
- we will run it on QEMU (hw emulator)
- a simpler version of the early linux kernel

Summary of Lab 1

- learn to run xk and debug using GDB
- read existing code and understand existing design decisions
- implement file syscalls
  - parsing and validating syscall arguments
    - see implemented syscalls for reference (sysfile.c)
    - argptr, argstr, argint, what do these functions do?
  - create open file (I/O) abstraction
    - user: file descriptor
    - kernel: file_info, file_* functions
  - perform the requested file operations
    - use the existing xk filesys (kernel/fs.c)

List of Syscalls To Support

`open(filename)`
returns a per-process handle (file descriptor) to be used in subsequent calls

`dup(fd)`
allocates a new file descriptor for the open file mapped by the `fd`

`close(fd)`
closes/deallocates a file descriptor

`read/write(fd, buffer, bytes_requested)`
reads or writes bytes into/out of buffer, advances position in file

`fstat(fd,stat)`
populates stat struct with information of the open file mapped by the `fd`

File Descriptors - User View

- implemented as an integer
- used for all I/Os
    - network sockets
    - pipes for interprocess communication
    - applications can use read/write regardless of what it is reading/writing to
- per-process construct
    - the same fd can map to different open files in different processes
- Kernel *should not* trust file descriptors passed by user
    - what could go wrong?

File Descriptors - Kernel View

- kernel allocates a file descriptor upon an open or dup
  - must be give out the smallest available fd
  - need to manage fd allocation
    - where might you store fd => open file mappings?
  - there's a max number (`NOFILE`) of open files for each process
    - what should happen if a process try to open more files?
- kernel deallocates file descriptors upon close
  - close(1) means that fd 1 is now available to be recycled and given out via open

The Open File Abstraction: File Info Struct

Needed to support richer semantics than what the xk filesys currently provides:

- the same file can be opened in different modes
- implicit file position advancement
- multiple fds can map to the same open file
- allocation & deallocation of the open file

File Info Struct

The Open File Abstraction: File Info Struct

What info do we need to support these semantics?

- reference count of the struct
  - how many fds points to this open file (why is this important?)
- a pointer to the inode of the file
- current offset of the open file
- access mode (check out inc/fcntl.h)
- anything else?

File Info Struct

Allocation of File Structs

After defining the file struct, you can pre-allocate `NFILE` amount of file info struct as a static array, and then actually allocate the struct when needed.

File_* Functions

Should implement a file_* for each of the file syscalls

File_* functions should take care of changes to the file info struct

- advancing the offset
- manage open file (file info struct) reference count
- allocate & deallocate struct when needed
- checking whether an operation is allowed given the access mode

The xk Filesys: Inode Layer

`iopen`

looks up a file using a given path, returns inode for the file
increments the inode's reference count

`irelease`

decrements this inode's reference count

`concurrent_readi`

read data using this inode

`concurrent_writei`

write data using this inode

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

Note: For Lab 1, don't worry about what inode is, just need to invoke the corresponding func.

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