Week 5
C Debugging

4 February 2019

CS 35L Lab 4

Jeremy Rotman

Announcements

- Assignment #4 is due Saturday by 11:55pm
- For Assignment #10
  - **Email me to tell me what story you are choosing**
  - Here is the link to see what stories people have signed up for already
    - Choose a story at least one week before you present
  - Here is the link to sign up to present
    - If you haven't signed up do it
  - I need article decisions from Max and Sriram

Questions?

Outline

- ➔ Debugging
- ➔ GDB
- ➔ Lab 4

Debugging

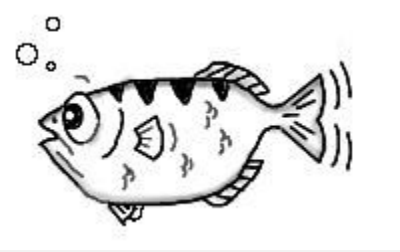
Debugging

- Finding and eliminating errors from programs
- Process
  - Reproduce the bug
  - Simplify input
  - Use a debugger to track down the origin of the problem
  - Fix the problem

Debugger

- Program that can run and debug other programs
- Advantages
  - Step through the code line by line
  - Interact with and inspect the code at run-time
  - More informational output when the program crashes

GDB



- GNU Debugger
- Allows debugging for multiple languages
  - C, C++, Assembly, and more
- Already built in and usable
- Helps us primarily to find segmentation faults and logical errors
- https://www.cs.cmu.edu/~gilpin/tutorial/
  - I'm using some examples from this tutorial, which includes simple code to try it yourself

GDB

- Compiling for GDB
  - `gcc [other flags] -g src.c -o src`
  - The -g option links debugging symbols into the program
    - This makes GDB more effective
- Entering GDB
  - Now that you have your executable, you can enter the debugger with it
  - **gdb** *src*
  - OR
    - **gdb**
    - (gdb) **file** *src*

GDB

- Running the Executable
  - `(gdb)` **`run`** OR
  - `(gdb)` **`run`** `[`*`arguments`*`]`
- Within the Interactive Shell
  - Tab autocompletes
  - Up-down arrows to scroll history
  - **`help`** `[`*`command`*`]`
    - Get more info on command
- Exit the debugger
  - `(gdb)` **`quit`**

Run-Time Errors

- ➔ Segmentation Fault

Run-Time Errors

- ➔ Segmentation Fault
  - ◆ Usually the result of trying to read or write an illegal memory location
  - ◆ When run without GDB, we normally just get the following error
    - Segmentation Fault
  - ◆ GDB gives us a different report:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400fee in Node<int>::next (this=0x0) at main.cc:29
29     Node<T>* next () const { return next_; }
```

    - This gives you the line number it crashed at along with the parameters to the function call that caused the crash

Backtrace

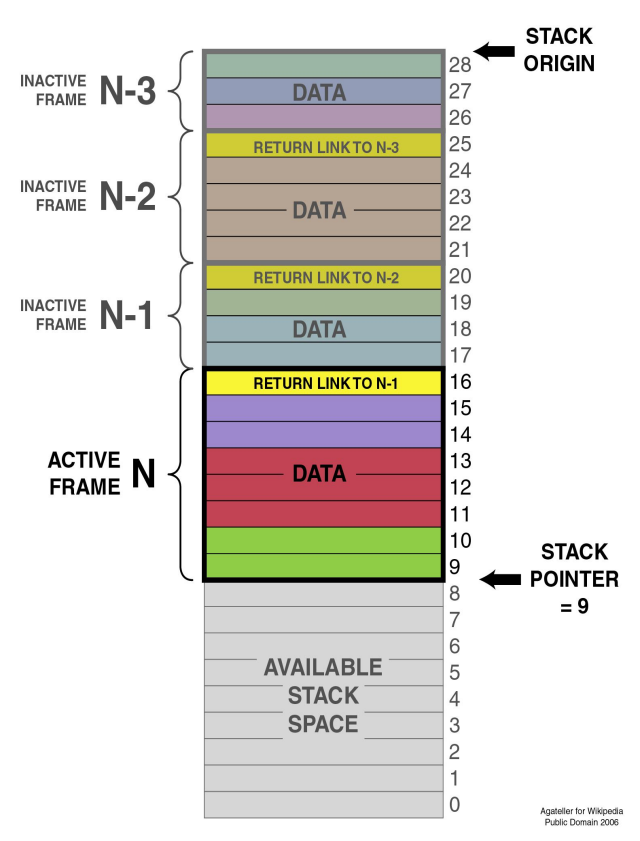
- **Backtrace** tells you more information about the point at which the program crashed.

```
#0  0x0000000000400fee in Node<int>::next (this=0x0) at main.cc:29
#1  0x0000000000400f06 in LinkedList<int>::remove (this=0x602010,
    item_to_remove=@0x7fffffffe23c: 1) at main.cc:78
#2  0x0000000000400bc9 in main (argc=1, argv=0x7fffffffe348) at
main.cc:121
```

- This gives us the path of function calls that lead to the specific call causing the fault

Backtrace

- Specifically this is information about the stack frame for each function
  - Stack frame: the space in memory set aside for a function
    - Stores all local variables
    - Stores memory address to return to when the called function returns
    - Stores the arguments of the called function
- When each stack frame is stacked atop each other, it becomes the call stack



Analyzing the Stack

- `(gdb) x mem_address`
  - This gives us the value being stored at that space in memory
- `(gdb) info frame`
  - Displays info about the current stack frame
- `(gdb) info locals`
  - Displays the local variables of the stack frame's function
- `(gdb) info args`
  - Displays the argument values of the function call

GDB

➔ Does this cover everything we might want to know?

GDB

- ➔ Does this cover everything we might want to know?
  - ◆ What if we have a logical error?
    - There will be no segfault to stop the program from “successfully” finishing
  - ◆ What if we want to stop before the segfault?
    - Maybe we want to look at the other function calls that didn’t segfault

Breakpoints

Breakpoints

- ➔ Allows you to specify a point in the code to stop running and wait for additional user input
- ➔ There are multiple ways to define a breakpoint

Breakpoints

- Allows you to specify a point in the code to stop running and wait for additional user input
- There are multiple ways to define a breakpoint
  - `(gdb) break main.cc:29`
    - Tells gdb to break at that specific line

Breakpoints

- ➔ Allows you to specify a point in the code to stop running and wait for additional user input
- ➔ There are multiple ways to define a breakpoint
  - ◆ `(gdb) break main.cc:29`
    - Tells gdb to break at that specific line
  - ◆ `(gdb) break LinkedList<int>::remove`
    - Tells gdb to break every time the function remove is called

Breakpoints

- ➔ Allows you to specify a point in the code to stop running and wait for additional user input
- ➔ There are multiple ways to define a breakpoint
  - ◆ `(gdb) break main.cc:29`
    - Tells gdb to break at that specific line
  - ◆ `(gdb) break LinkedList<int>::remove`
    - Tells gdb to break every time the function remove is called
  - ◆ `(gdb) break LinkedList<int>::remove if items_to_remove==1`
    - Tells gdb to break if the function remove is called with the parameter items_to_remove being equal to 1.

Breakpoints

- ➔ Once set, you can rerun the program and it will stop when it reaches the breakpoint
- ➔ You can have many breakpoints
  - ◆ `(gdb) info breakpoints | break | br | b`
    - Shows the list of breakpoints and information about where they are set
    - Will also include the breakpoint number which is important for other commands that deal with breakpoints

Breakpoints

- `(gdb)` **`delete`** *`bp_number`*
  - Deletes the breakpoint, or range of breakpoints if given range
- `(gdb)` **`disable`** *`bp_number`*
  - Disables, but doesn't remove, the breakpoint(s)
- `(gdb)` **`enable`** *`bp_number`*
  - Enables previously disabled breakpoint(s)
- If no arguments are given, it affects all breakpoints
- `(gdb)` **`ignore`** *`bp_number iterations`*
  - Pass over a breakpoint without stopping *iterations* number of times

Displaying Data

- If you know what variables should exist at the breakpoint
  - You can display what value they hold
- `(gdb) print [/format] expression`
  - Prints the value of expression in the desired format
- Format options
  - d: decimal notation
  - x: hexadecimal notation
  - o: octal notation
  - t: binary notation
- Default is to use whatever notation is most appropriate for the data type, decimal for integers

After the Breakpoint

- ➔ There are 4 operations to continue after the breakpoint
  - ◆ **c** or **continue**
    - Debugger will continue to run until it hits another breakpoint, error, or finishes running

After the Breakpoint

- There are 4 operations to continue after the breakpoint
  - **c** or **continue**
    - Debugger will continue to run until it hits another breakpoint, error, or finishes running
  - **s** or **step**
    - Debugger will continue to next line
    - Steps into function calls

After the Breakpoint

- There are 4 operations to continue after the breakpoint
  - **c** or **continue**
    - Debugger will continue to run until it hits another breakpoint, error, or finishes running
  - **s** or **step**
    - Debugger will continue to next line
    - Steps into function calls
  - **n** or **next**
    - Debugger will continue to next line in current stack frame
    - Steps over function calls

After the Breakpoint

- There are 4 operations to continue after the breakpoint
  - **c** or **continue**
    - Debugger will continue to run until it hits another breakpoint, error, or finishes running
  - **s** or **step**
    - Debugger will continue to next line
    - Steps into function calls
  - **n** or **next**
    - Debugger will continue to next line in current stack frame
    - Steps over function calls
  - **f** or **finish**
    - Debugger will continue until the current function returns to the function that called it

Watchpoints

- Watchpoints are attached to variables and will stop the program whenever an action is taken on the variable
- `(gdb)` **`watch`** *`expression`*
  - Set watchpoint on *expression*
  - Stop when the value of *expression* is changed
  - Prints old and new values
- `(gdb)` **`rwatch`** *`expression`*
  - Stops when the value of *expression* is read

Additional Commands

- `(gdb) info functions`
  - Lists all functions in the program
- `(gdb) list`
  - Lists source code lines around the current line

Lab 4

- More Coreutils!
- We are now working with a different, though still old, version of coreutils
- ls has a bug, again
- You will first attempt to build coreutils as-is
  - Then after will build it with a renaming patch
- Then reproduce the error with ls, and use the debugger to figure out how to fix it
  - You will be submitting a patchfile to fix the bug
- You will also need to submit your log of what you did

Lab 4 Hints

- Attempt to simplify the input by using the flags l and t individually
- To run GDB on the local ls
  - `gdb ./ls`
  - `(gdb) run -lt wwi-armistice now now1`
- Make sure to answer all questions in lab4.txt
- Make sure your patch is in the correct direction
  - Outlines how to change from the old version to the fixed version
- Use info functions to look for a good starting point for debugging

Questions?