Software Construction Laboratory

Week 4 Part 2

Lab 3

Mushi Zhou

Winter 2017

Prof. Eggert

UCLA

Outline

- Debuggers
- Debugging tools
- GDB
- Valgrind
- Strace

Debugging

- Debugging is the process of finding and resolving of defects that prevent correct operation a program
- It is not always a trivial thing

Two types of errors

- Syntax & Logic
- Debugging syntax errors are usually easy, just locate the error with the report and correct it
- Logic errors are run time errors, meaning the code compiles but behave differently than expected
- Logic errors could be non-obvious or hidden
- Unfortunately, C doesn’t provide too much information when crushes with memory issues, other than Segmentation Fault in many cases
- Need tools to help find out what went wrong

Introduction to GDB

- The GNU project debugger, allows you to see what is going on “inside” another program while it executes
- Start your program, specifying anything that might affect its behavior
- Make your program stop on specified conditions
- Examine what has happened, when your program has stopped
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another
- Version 7.12 released this month, October 2016

GDB Commands

Basic commands

- gdb programName // Open GDB with the program to debug
- run –v args // Execute the program in GDB (not automatic)
  // with arguments
- bt // Show back trace results if the program crushes
- print var // Print the current value of variable var,
  // same as p var

Displaying Source Code within GDB Session

- list [*filename*:] *line_number*
  - Displays source code centered around the numbered line
  - For single source code file program, you don’t need to worry about filename
- list *from*,[*to*]
  - Displays the specified range of the source code
  - Can be either line numbers or function names
- list *function_name*
  - Displays source code centered around the line in which the specified function begins.

Breakpoint

- In order to view what is happening in the middle of an execution, breakpoint is essential
- break
  - Sets a breakpoint at the next statement in the same stack frame to be executed. In other words, the program flow will be automatically interrupted when the execution reaches the next statement right below the current statement
- break *line_number*
  - Sets a breakpoint at the specified line
- break [*filename:]functionname*
  - Sets a breakpoint at the first line of the specified function
  - If the function is in a different you can specify the file name

Conditional Breakpoints

- break [*line_number*] if *expression*

For example:

->27 for (i = 1; i <= limit ; ++i)

(gdb) break 28 if i == limit - 1

Breakpoint 1 at 0x4010e7: file gdb_test.c, line 28.

// The scope of all variables are local to the gdb commands

// Each breakpoint is assigned a number as displayed

Deleting, Disabling, and Ignoring BP

- d *bp_number*
  - Deletes the specified breakpoint
  - A delete command with no argument deletes all the breakpoints that have been defined
  - GDB tells you the bp_number associate with your breakpoint when you create them
- enable, disable works the same way
- ignore *bp_number #times*
  - Instructs GDB to pass over a breakpoint without stopping a certain number of times
  - The ignore command takes two arguments: the number of a breakpoint, and the number of times you want it to be passed over
- continue (c) // Execute until next breakpoint reached
- **step (s) // Execute the next line -> may be in another function or file that the current line leads to**
- **next (n) // Execute until after the next line the current function or stack frame, i.e. the next line below**
- finish // Execute until the current function finishes

More on GDB

- GDB has a lot more advanced commands: http://www.yolinux.com/TUTORIALS/GDB-Commands.html

- Examples: https://www.tutorialspoint.com/gnu_debugger/gdb_debugging_example1.htm

Valgrind

- Valgrind is a programming tool for memory debugging, memory leak detection, and profiling

Can show:

- Use of uninitialized memory
- Reading/writing memory after it has been free'd
- Reading/writing off the end of malloc'd blocks
- Occupied memories allocated but not free’d at program exit
- Memory leaks -- where pointers to malloc'd blocks are lost forever
- And more

Valgrind Example

valgrind --tool=memcheck --leak-check=yes ./program

- This outputs a report to the terminal including:
- ==9704== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 11 from 1)
- ==9704== malloc/free: in use at exit: 35 bytes in 2 blocks.
- ==9704== malloc/free: 3 allocs, 1 frees, 47 bytes allocated.
- ==9704== checked 1420940 bytes.
- ==9704== 16 bytes in 1 blocks are definitely lost in loss record 1 of 2
- ==9704== at 0x1B903D38: malloc (vg_replace_malloc.c:131)
- ==9704== by 0x80483BF: main (test.c:15)

Valgrind Example Continued

- ==9704== 19 bytes in 1 blocks are definitely lost in loss record 2 of 2
- ==9704== at 0x1B903D38: malloc (vg_replace_malloc.c:131)
- ==9704== by 0x8048391: main (test.c:8)
- ==9704== LEAK SUMMARY:
- ==9704== definitely lost: 35 bytes in 2 blocks.
- ==9704== possibly lost: 0 bytes in 0 blocks.
- ==9704== still reachable: 0 bytes in 0 blocks.
- ==9704== suppressed: 0 bytes in 0 blocks.

Other tools

- Strace, a diagnostic and debugging tool for Linux which show useful information beween processes and the kernal, i.e. system calls (We will talk more about this in week 8)
- LLDB, a new debugger that can debug C, Objective-C, C++ and Swift
- WinDbg, a windows system debugger
- etc.

Some Help for Assignment 4

For the lab

- Think about how system stores timestamp (in what type)
- If there is a base timestamp, and the given timestamp is earlier in the timeline than the base timestamp, how does it affect the stored timestamp?
- Do you see your touched time become a time in the far future?

Clarifications for Assignment 5

- Overflow? Underflow?
- Fix should be simple
- GDB is not a magic thing that automatically tells you what went wrong. It is a tool to help you find the problem by hand
- You might want to inspect the source code and roughly locate where the problem might be and then use GDB to inspect the execution
- You need to give arguments to GDB to run your program just like in command line in order for execute the portion of the program you want it to

Some Help for Assignment 5

For HW5

- You don’t want to use memfrob, memcmp, but its helpful to understand them
- SPACE and new line “\n” are switched in memfrob translation
- Qsort takes a pointer to the comparison function (frobcmp in this case)
- What functions do you use to read input? (You are reading from stdin)
- You want to dynamically adding memory allocations as you read since you don’t know the total size in the first place
- How do you order two bytes?
- The input file could be empty