

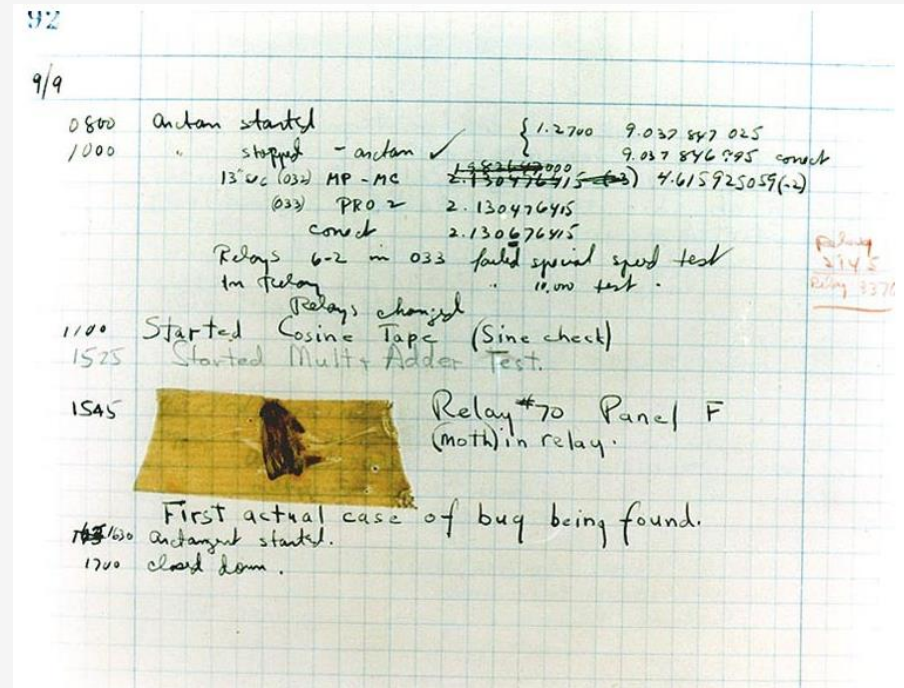
CS35L – Winter 2019

Slide set:	4.2
Slide topics:	GDB, debugging
Assignment:	4



Debugging

- Finding and eliminating errors from programs
- Grace Hopper and the “First actual case of bug being found”



Debugging Process



- Reproduce the bug
 - Simplify program input
 - Use a debugger to track down the origin of the problem
 - Fix the problem
-

Debugger

- A program that is used to run and debug other (target) programs
- Advantages:
 - Programmer can:
 - step through source code line by line
 - each line is executed on demand
 - interact with and inspect program at run-time
 - If program crashes, the debugger outputs where and why it crashed

GDB – GNU Debugger



Debugger for
several
languages

C, C++, Java,
Objective-C, etc.



Allows you to inspect
what the program is
doing at a certain point
during execution



Logical errors and
segmentation faults
are easier to find with
the help of gdb



Using GDB

1. Compile Program

- Normally: `$ gcc [flags] <source files> -o <output file>`
- Debugging: `$ gcc [other flags] -g <source files> -o <output file>`
 - enables built-in debugging support

2. Specify Program to Debug

- `$ gdb <executable>`
- or
- `$ gdb`
- `(gdb) file <executable>`



Using GDB

3. Run Program

- (gdb) run or
- (gdb) run [arguments]

4. In GDB Interactive Shell

- Tab to Autocomplete, up-down arrows to recall history
- help [command] to get more info about a command

5. Exit the gdb Debugger

- (gdb) quit

Run-Time Errors vs Logical Errors

- Runtime Error/Segmentation fault
 - Program received signal SIGSEGV, Segmentation fault. 0x0000000000400524 in *function* (arr=0x7fffc902a270, r1=2, c1=5, r2=4, c2=6) at *file.c*:12
 - Line number where it crashed and parameters to the function that caused the error
- Logical Error
 - Program will run and exit successfully
- How do we find bugs?

Setting Breakpoints

- Breakpoints
 - used to stop the running program at a specific point
 - If the program reaches that location when running, it will pause and prompt you for another command
- Example:
 - (gdb) break file1.c:6
 - Program will pause when it reaches line 6 of file1.c
 - (gdb) break my_function
 - Program will pause at the first line of my_function every time it is called
 - (gdb) break [*position*] if *expression*
 - Program will pause at specified position only when the expression evaluates to true

Breakpoints



- Setting a breakpoint and running the program will stop program where you tell it to
- You can set as many breakpoints as you want
 - (gdb) info breakpoints | break | br | b shows a list of all breakpoints

Deleting, Disabling and Ignoring BPs

- (gdb) delete [bp_number | range]
 - Deletes the specified breakpoint or range of breakpoints
- (gdb) disable [bp_number | range]
 - Temporarily deactivates a breakpoint or a range of breakpoints
- (gdb) enable [bp_number | range]
 - Restores disabled breakpoints
- If no arguments are provided to the above commands, all breakpoints are affected!!
- (gdb) ignore bp_number iterations
 - Instructs GDB to pass over a breakpoint without stopping a certain number of times.
 - bp_number: the number of a breakpoint
 - Iterations: the number of times you want it to be passed over



Displaying Data

- Why would we want to interrupt execution?
 - to see data of interest at run-time:
 - (gdb) `print [/format]
expression`
 - Prints the value of the specified expression in the specified format
 - Formats:
 - d: Decimal notation (default format for integers)
 - x: Hexadecimal notation
 - o: Octal notation
 - t: Binary notation



Resuming Execution After a Break

- When a program stops at a breakpoint
 - 4 possible kinds of gdb operations:
 - **c or continue**: debugger will continue executing until next breakpoint
 - **s or step**: debugger will continue to next source line
 - **n or next**: debugger will continue to next source line in the current (innermost) stack frame
 - **f or finish**: debugger will resume execution until the current function returns. Execution stops immediately after the program flow returns to the function's caller
 - the function's return value and the line containing the next statement are displayed

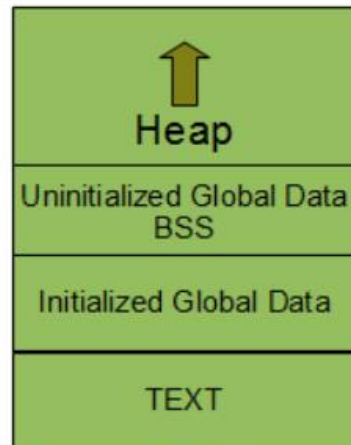
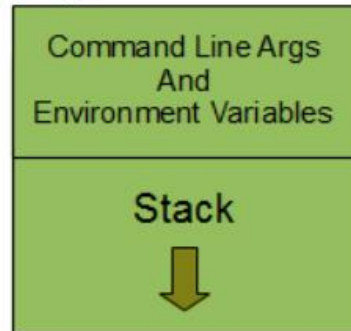
-
- Watch/observe changes to variables
 - (gdb) `watch my_var`
 - sets a watchpoint on `my_var`
 - the debugger will stop the program when the value of *my_var* changes
 - old and new values will be printed
 - (gdb) `rwatch expression`
 - The debugger stops the program whenever the program reads the value of any object involved in the evaluation of *expression*

Watchpoints



Process Memory Layout

(Higher Address)




(Lower Address)

- TEXT segment
 - Contains machine instructions to be executed
- Global Variables
 - Initialized
 - Uninitialized
- Heap segment
 - Dynamic memory allocation
 - malloc, free
- Stack segment
 - Push frame: Function invoked
 - Pop frame: Function returned
 - Stores
 - Local variables
 - Return address, registers, etc
- Command Line arguments and Environment Variables

-
- A program is made up of one or more functions which interact by calling each other
 - Every time a function is called, an area of memory is set aside for it. This area of memory is called a **stack frame** and holds the following crucial info:
 - storage space for all the local variables
 - the memory address to return to when the called function returns
 - the arguments, or parameters, of the called function
 - Each function call gets its own stack frame. Collectively, all the stack frames make up the **call stack**

Stack Info



Analyzing the Stack in GDB

- (gdb) backtrace|bt
 - Shows the call trace (the call stack)
 - Without function calls:
 - #0 main () at program.c:10
 - one frame on the stack, numbered 0, and it belongs to main()
 - After call to function display()
 - #0 display (z=5, zptr=0xbffffb34)
at program.c:15
 - #1 0x08048455 in main () at
program.c:10
 - Two stack frames: frame 1 belonging to main() and frame 0 belonging to display().
 - Each frame listing gives
 - the arguments to that function
 - the line number that's currently being executed within that frame

Analyzing the Stack

- `(gdb) info frame`
 - Displays information about the current stack frame, including its return address and saved register values
- `(gdb) info locals`
 - Lists the local variables of the function corresponding to the stack frame, with their current values
- `(gdb) info args`
 - List the argument values of the corresponding function call



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

Other Useful Commands



- Download old version of coreutils with buggy ls program
 - Untar, configure, make
- Bug: ls -t mishandles files whose time stamps are very far in the past. It seems to act as if they are in the future

```
$ tmp=$(mktemp -d)
```

```
$ cd $tmp
```

```
$ touch -d '1918-11-11 11:00 GMT' wwi-armistice
```

```
$ touch now
```

```
$ sleep 1
```

```
$ touch now1
```

```
$ TZ=UTC0 ls -lt --full-time wwi-armistice now now1
```

```
-rw-r--r-- 1 eggert csfac 0 1918-11-11 11:00:00.000000000 +0000 wwi-  
armistice
```

```
-rw-r--r-- 1 eggert csfac 0 2017-01-25 00:11:55.528846902 +0000 now1
```

```
-rw-r--r-- 1 eggert csfac 0 2017-01-25 00:11:54.524820127 +0000 now
```

```
$ cd
```


```
$ rm -fr $tmp
```

- **Reproduce the Bug**
 - Follow steps on lab web page
- **Simplify input**
 - Run ls with -l and -t options only
- **Debug**
 - Use gdb to figure out what's wrong
 - `$ gdb ./ls`
 - `(gdb) run -lt /tmp/wwi-armistice /tmp/now /tmp/now1`

(run from the directory where the compiled ls lives)
- **Patch**
 - Construct a patch "lab4.diff" containing your fix
 - It should contain a ChangeLog entry followed by the output of diff -u

*Goal: Fix
the Bug*

Lab Hints

- Don't forget to answer all questions! (lab4.txt)
 - Make sure not to submit a reverse patch! (lab4.diff)
 - "Try to reproduce the problem in your home directory, instead of the \$tmp directory. How well does SEASnet do?"
 - Timestamps represented as seconds since Unix Epoch
 - SEASnet NFS filesystem has unsigned 32-bit time stamps
 - Local File System on Linux server has signed 32-bit time stamps
 - If you touch the files on the NFS filesystem it will return timestamp around 2054
 - => files have to be touched on local filesystem (df -l)
 - Use "info functions" to look for relevant starting point
- 
-