ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Debugging

- Even when written in high-level languages, programs have bugs
  - Recall the thought that when moving away from assembly bugs would disappear!
- Some famous bugs
  - Mariner I Venus probe (1962)
    - Had to be destroyed as it went off course (wrong loop? wrong cut-and-paste?)
  - Ariane 5 failure (1996)
    - Some variables changed from int to long, some not...

Debugging

- Programmers and debugging
  - Some people love debugging
    - Sense of accomplishment
  - Some people hate it
    - Difficult, and not really taught
- Debugging: *determining the exact nature and location of a suspected error and fixing it*
  - Locating the error is often 95% of the work
- Question: how do we find bugs?
- Two main approaches:
  - Static Debugging
    - Visual Inspection
    - Fancy name for all types of “monkeying around” with the code
  - Dynamic Debugging
    - Using a debugger

Static Debugging

- Stare at the code
  - Has its limits, as we know
    - Although asking a peer for code review can work better
  - Using all types of compiler flags so that it generates all possible warnings is a good idea
    - e.g., gcc -Wall -pedantic
- Puts a bunch of printf statements
  - "I'm here", "I am seeing this value"
  - Widely used and pretty effective
  - But can be labor-intensive
    - Both to instrument the code and to look at the output
- Comment-out portions of the code
  - To find compilation errors mostly
- Stare at the code again
  - But not too passively!!!

Static Debugging: Limitations

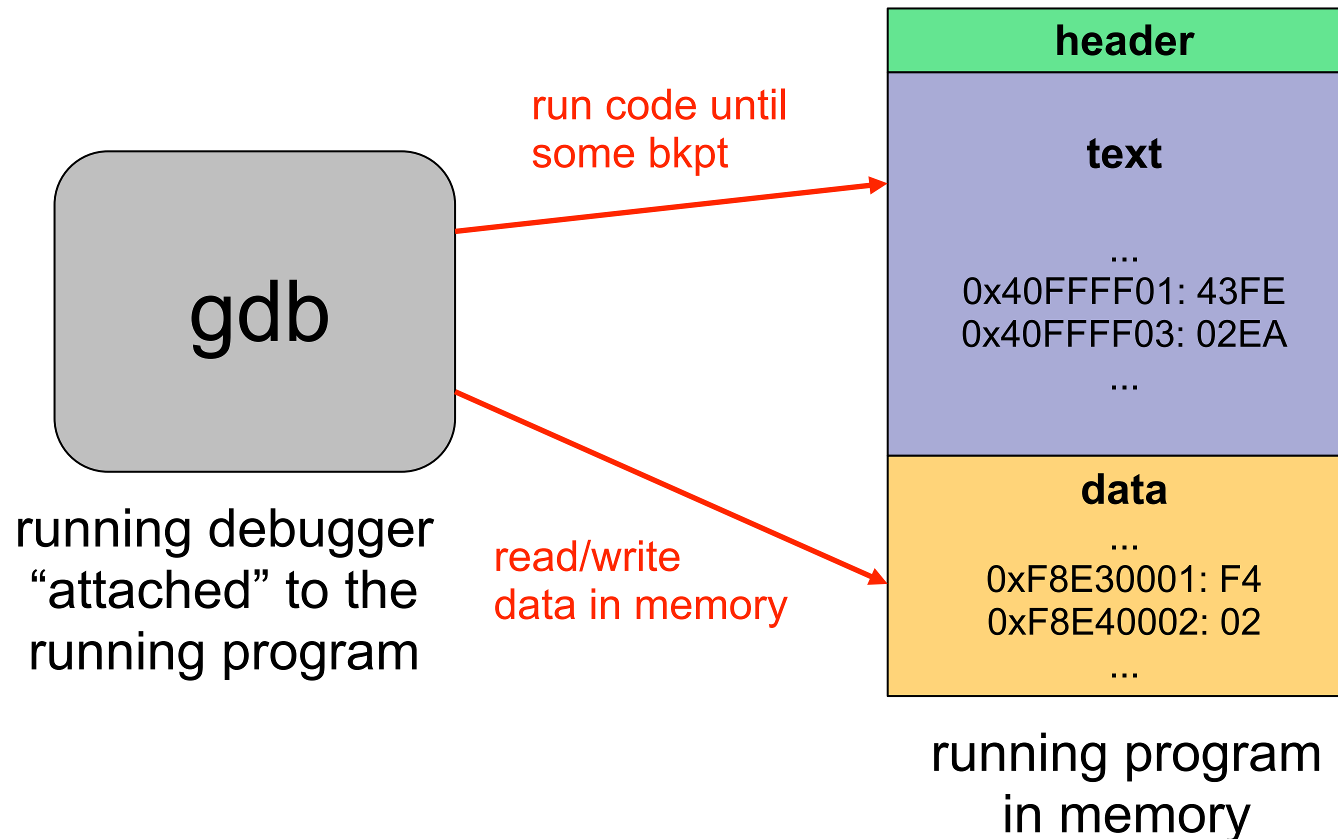
- The problem with all static debugging techniques is that there are things you almost cannot do:
  - What about code in libraries that you use but didn’t write?
  - What about memory bugs in languages like C?
- For these above limitations, and the ones seen on the previous slides, we have dynamic debugging

Dynamic Debugging

- Dynamic debugging makes it possible to observe a program as it runs
  - And to control its execution
- To do this we use a debugger
- The principles of a debugger
  - You compile the code enabling debugging
    - Debugging information is embedded inside the object files, so that the debugger can relate machine code to high-level code
  - You run your code within the debugger
- The debugger allows you to:
  - Run step-by-step
  - Insert breakpoints
  - Look at variable contents
  - Modify variable contents

The GNU Debugger

- The GNU Debugger is gdb

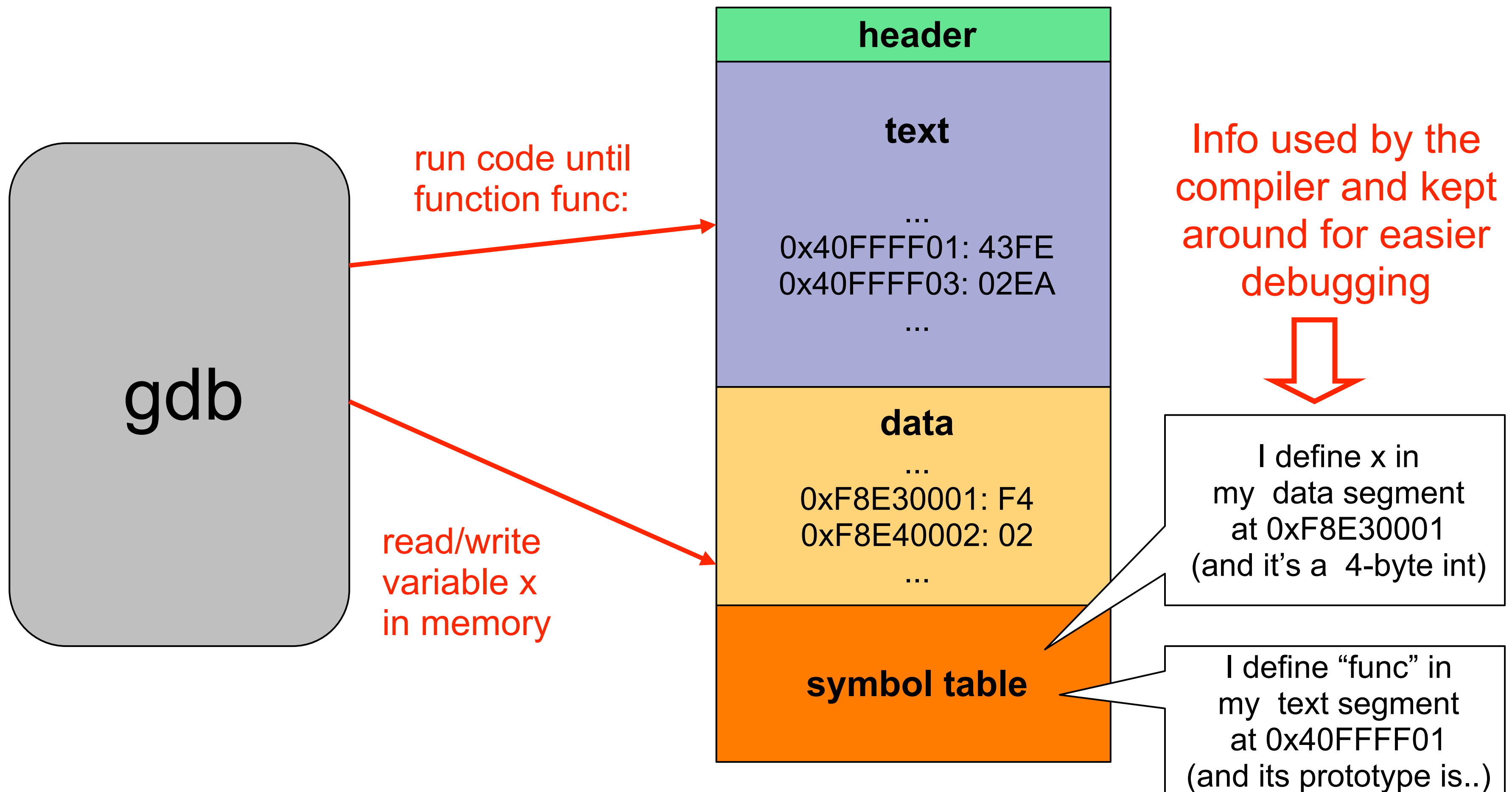


Including a Symbol Table

- The problem with the previous picture is that the everything is binary (or hex, which is only a little bit more readable)
- So as a user of the debugger, if you want to look at the content of a variable, you have to specify its address:
  - “Tell me the 2-byte value at 0xFFE40123”
- What we would really like to do is only use variable names as declared/used in the program
  - “Tell me the value of variable x”
- To do so, we must have a symbol table
  - Recall the linker/loader lecture
- In general the symbol table is removed from the final executable
  - Takes space and isn’t used for running the program
- You can tell gcc to keep it around by compiling with the -g flag:
  - gcc -c **-g** main.c -o main.o

The GNU Debugger

- The GNU Debugger is gdb



Running gdb

- To run your program under the control of the debugger you just invoke is as follows:
  - gdb ./prog
- At this point we are within the gdb prompt and we can type gdb commands
- Let's try this and look at the gdb ever useful "help" command
  - On a Linux box....

Useful gdb Commands

- run (or ‘r’): starts the program
  - with potential command-line arguments
  - the program will run all the way through
- list (or ‘l’): shows 10 lines of code around “where we are”
- break (or ‘b’): sets a breakpoint
  - In a function, e.g., “break main”
  - At a specific line in the code, e.g., “break main.c:154”
- step (or ‘s’): runs the program step-by-step
  - After stopping at a breakpoint
- next (or ‘n’): like step, but skips over functions
- continue (or ‘c’): continues until next breakpoint
- print (or ‘p’): to print variable values
  - or function call!
- quit (or ‘q’): quits the program/debugger

Managing Breakpoints

- To list all existing breakpoints you can use the info break command
  - Example:

```
(gdb) info break

Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x001f7c in main at main.c:4
2   breakpoint     keep y   0x001f96 in main at main.c:12
3   breakpoint     keep y   0x001fa9 in main at main.c:17
```

- To delete a breakpoint you can use the delete command
  - Example: delete 2

Post-Mortem debugging?

- A very important use of a debugger is when the program simply causes a segmentation fault or a bus error
  - Meaning, there is something fishy that happens with the use of the memory
  - Especially useful for languages like C and C++ of course
- Note that such errors are very rare in a language like Java
  - So much “hand holding” of the developer
- The program has already crashed so we can’t observe the bug “live”

Post-Mortem with Cores

- When a program encounters a fatal error, the Kernel terminates execution and creates a “core file” (a.k.a. “core dump”)
  - A core file is a snapshot of the program at the moment the error occurred
- This happens upon
  - Segmentation faults: trying to access memory that is not in the address space of the process
  - Bus errors: (often) trying to address an address that cannot be physically addressed
  - Illegal instruction: happens when execution branches into data
  - Arithmetic exception: e.g., dividing by zero
- Once a core file has been generated, you can invoke gdb as:
  - gdb <prog name> <core file>

Bells and Whistles…

- gdb has tons of cool options
- One example: conditional breakpoints
  - Say you want a breakpoint to stop the code only if a certain condition is met
    - So that you don’t have to type continue a bunch of times until you get the code in the place you want (with the risk of missing it!)
  - You can just type something like:
    - cond 3 x > 100
  - which will make breakpoint #3 stop only if variable x (in the context of the breakpoint) is strictly larger than 100!

So now what?

- At this point, we can do a detail step-by-step run through the code and inspect all relevant memory content
- But what about the most vexing bugs (in C): memory corruption!
  - going over the end of an array
  - writing the memory that was freed
- Memory corruption is very tricky to debug
  - It may cause the program to give a wrong result, without a segfault
  - A segfault may be caused by an instruction that is not the one that corrupted memory
- Using printf statements to find memory corruption bugs is hopeless
  - adding printf’s can appear to “magically” fix the memory bug!
- Using a debugger is also quite difficult
  - One has to look at all memory content...

Memory Bugs?

- Memory bugs in languages like C can be very hard to fix
- What we really need is an automatic way in which memory integrity can be checked
- A popular opensource/free tool is valgrind
- You simple run your code through valgrind as:
  - valgrind ./prog [arguments]
- And then you look at the valgrind report

Conclusion

- Using print statements to debug code is extremely limited
  - Of course when writing assembly we have used it, but with high-level code it’s really cumbersome
- When you start having many data structures, especially in a language like C, you have to use more powerful tools
  - A debugger, like gdb
  - A memory checker, like valgrind
- Somehow, the temptation to not use them is great!
  - But one should not succumb to it, especially when facing memory issues