Debugging

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

There will be bugs…

- Even when written in high-level languages, programs have bugs
  - Recall the thought that when moving away from assembly, in the 50's, 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…

There should be testing…

- One way to find bugs is let your users report them
  - This will always happen, but if it's too much, you're toast
- The way to find bugs is to write testing code
- This is often seen as "boring", but it's the only way
- Often, your test code is much larger than your "real" code
- For instance, in my latest open-source project for my research, looking at .cpp files:
  - 17,475 lines of C++ for testing (with Google test)
  - 17,022 lines of C++ for the real code!!!
- This is pretty typical…..

Firefox

- 996,214 commits by 6,495 people
- 36M lines of code (22% of which are comments)

- Such projects have an amazing testing infrastructure, which takes as much or more effort than the code itself
  - Which consists of millions of lines of code, over dozens of projects

- So this is about finding bugs before users find bugs!
- And there are LOTS of bugs everywhere…

Lots of Bugs

Categorizing Bugs with Social Networks: A Case Study on Four Open Source Software Communities
Zanetti, Scholtes, Tessone, and Schweitzer

Table 1: Time periods, number of bugs, number of change events and number of bugs with particular status. The different bug resolution categories are the following: *FIX* for fixed, *DUP* for duplicate, *INV* for invalid, *WOF* for won't fix and finally *INC* for incomplete. More details in section 3.1.

	FIREFOX	THUNDERBIRD	ECLIPSE	NETBEANS	Total
Start date	April 2002	January 2000	October 2001	January 1999	–
Total bug reports	112,968	35,388	356,415	210,921	715,692
Change events	1,068,070	313,957	2,594,385	1,875,878	5,852,290
Changes / report	9.45	8.87	7.28	8.89	8.18
Resolved bugs (resolved/total)	64,088 (0.57)	21,644 (0.61)	158,957 (0.45)	42,851 (0.19)	287,540 (0.40)
FIX (FIX / resolved)	10,856 (0.17)	4,508 (0.21)	103,453 (0.65)	21,442 (0.50)	140,259 (0.49)
DUP (DUP / resolved)	24,263 (0.38)	10,336 (0.48)	28,227 (0.18)	9,328 (0.22)	72,154 (0.25)
INV (INV /resolved)	11,785 (0.18)	2,829 (0.13)	12,601 (0.08)	4,082 (0.10)	31,297 (0.11)
WOF (WOF / resolved)	2,708 (0.04)	581 (0.03)	14,676 (0.09)	5,515 (0.13)	23,480 (0.08)
INC (INC / resolved)	14,476 (0.23)	3,390 (0.16)	-	2484 (0.06)	20,350 (0.07)

Debugging

- Programmers and debugging
  - Some people love debugging
    - Sense of accomplishment
  - Some people hate it
    - Difficult, not really taught
    - 1 second of satisfaction before next bug
- Debugging: *determining the exact nature and location of a suspected error and fixing it*
  - Locating the error is often 95% of the work
  - But sometimes fixing one small bug means a whole redesign
- Question: how do we find bugs?
  - Static Debugging
    - Visual Inspection
    - Fancy name for all types of “monkeying around” with the code
  - Dynamic Debugging
    - Using a debugger

First Step: Compiler Flags!!

- It turns out some “bugs” can actually be caught by better use of compiler flags, in languages like C/C++
- Typically a great idea to use: gcc -Wall -pedantic
- Overkill options?
  - -Wshadow -Wcast-align -Wnonnull -Waggregate-return -Wmissing-prototypes -Wmissing-declarations -Wstrict-prototypes -Wredundant-decls -Wnested-externs -Wpointer-arith -Wwrite-strings -Wall
- Typically, your IDE does compilation on the fly with such flags, and shows you warnings visually
  - DO NOT IGNORE THESE!!
- IDEs also provide “code inspection” tools
  - RUN THESE!!

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 in “unsafe” languages like C
    - e.g., gcc -Wall -pedantic
  - Ignoring warning with a “it’s just a warning” attitude is BAD
- Puts a bunch of printf statements
  - “I’m here”, “I am seeing this value”
  - Widely used and pretty effective up to a point, depending on the language
- Comment-out portions of the code
  - To find compilation errors also (when compiler error messages are insane…. C++?)
  - Somehow seems not to occur to many students…
- Stare at the code again
  - But not too passively!!!

Static Debugging: Limitations

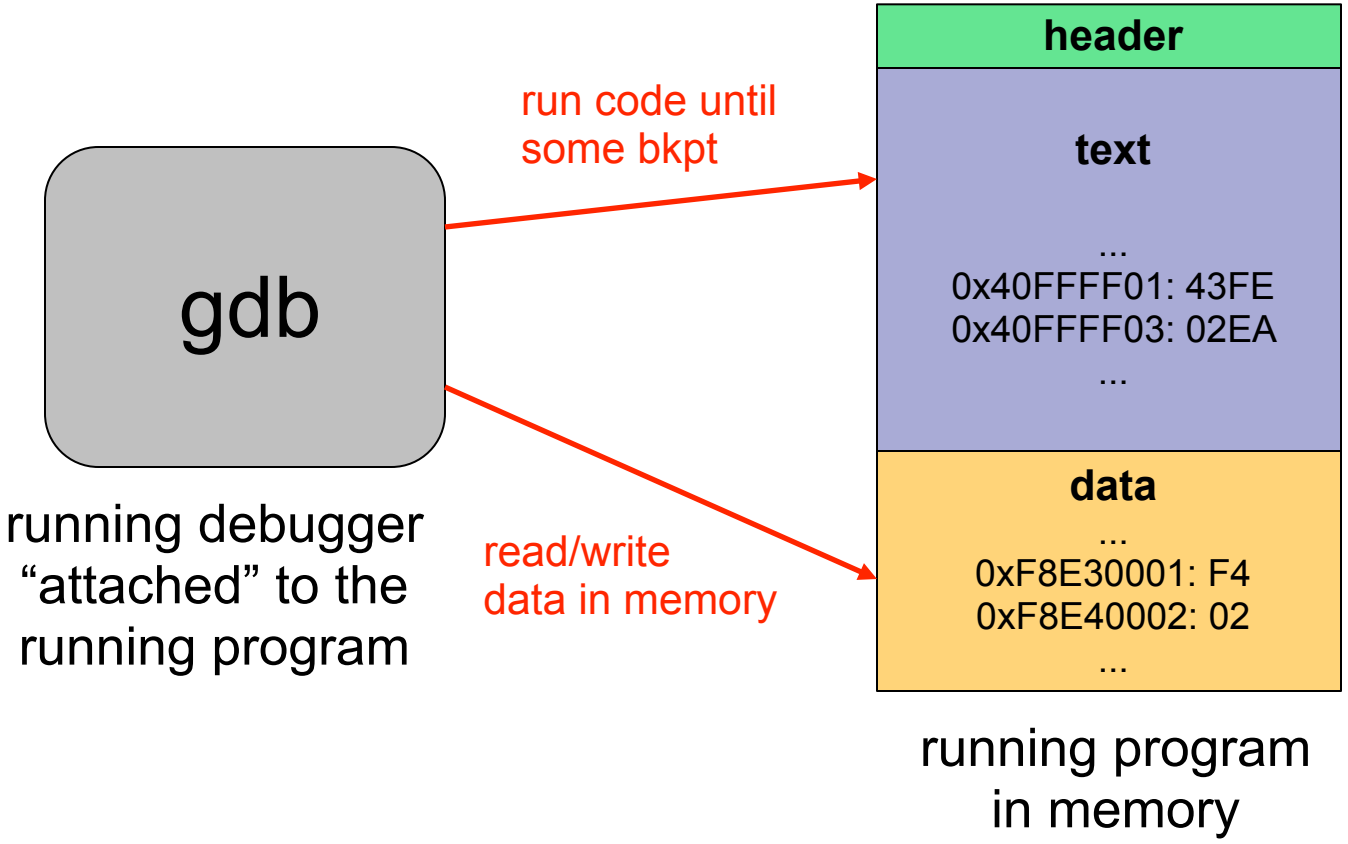
- The problem with all static debugging techniques is that there are things you cannot fix by staring/printing:
  - What about code in libraries that you use but didn't write?
  - What about memory bugs in languages like C?
- For these above limitations, we have dynamic debugging

Dynamic Debugging

- Dynamic debugging: observe a program as it runs and 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
    - We can see this with objdump and compiling with -g…
  - 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 (!!)
- How of hands: haver you used a debugger before? (poll)

The GNU Debugger: Low Tech

- 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 use the 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
- Compiling with the -g flag keeps it around (with human-readable symbol names):
  - gcc -c **-g** main.c -o main.o

IDE Debugger: High Tech

- These days, one typically runs fancy debuggers from the IDE
- They work on exactly the same principles and don’t have more capabilities
- But they’re just easier to use
- If the poll from “

Shows of Hands

- Show of hands #1: Have you used a debugger before? (poll)

- Show of hands #2: Do you want to go through an example of debugging a program with a debugger?
  - If so, let's run the debugger on code_example4.cpp

The Debugger is not Enough

- A debugger 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 is typically 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 at each steps

code_example5.c

- This is an example with a memory problem that would be labor-intensive to detect with a debugger...
- Is there an alternative?

Automatic Memory Checking

- What we really need is an automatic way in which memory integrity can be checked

- A popular open-source/free tool is valgrind
- You simple run your code through valgrind as:
  - valgrind ./prog [arguments]
- And then you look at the valgrind output
- Let's try it on code_example5.c to debug the program…

Memory Leaks are Bad

- Memory Leak: memory that's allocated (on the heap) but there are no longer any pointers to it
- **Question:** Why are Memory Leaks bad?
- **Answer:** As the program runs, its memory footprint increases

- For a short-lived program that doesn't use much RAM we don't care
  - But a higher memory footprint will likely slow down execution regardless (see ICS332)
- But for a long-running program, why will eventually run out of RAM and crash!
  - This is really bad for daemons/servers
- So the (good) programmer should avoid memory leaks
- How do we do that?

Removing Memory Leaks

- Some languages to automated Garbage Collection (Java, Python, etc)
  - Memory leaks cannot happen, we're done
- Other languages do not (C/C++): Memory needs to be deallocated
  - free, delete, etc.
- Approach #1: do it by hand
  - Be careful to free memory when it needs to be freed: can be really hard for complex program
- Approach #2: use "smart pointers" (e.g., C++)
  - Anybody knows what that is?
- Regardless, the programmers needs to be involved
- Which means Memory Leaks can occur

Using valgrind again

- Let’s try to make code_example5.c Memory Leak-free using valgrind…

Conclusion

- Heeding compiler warnings: paramount
- Staring at code and using print statements to debug: can quickly become very 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/C++, you have to use more powerful tools
  - A debugger
  - A memory checker, like valgrind
- For some reason, the temptation to not use tools and to "fake it" by hand is great
  - But one should not succumb to it because it is often a waste of time, especially when facing memory-related issues