Announcements

• Requirements due March 8th

Debuggers

CS169
Lecture 12

Outline

• Debugging
• How do interactive debuggers work?
  - A "real" debugger (gdb)
  - A research debugger

Debugging

• Occurs as a consequence of successful testing
• Testing is an orderly operation
  - Planned, systematic
• Debugging is more of an art
  - Map symptom to cause
  - Investigate suspected causes

"Everyone knows that debugging is twice as hard as programming; so, if you are as clever as you can be when you write the program, how will ever debug it?" — Kernighan

Why is Debugging So Hard?

• Symptom and cause can be remote
  - Statically and/or dynamically remote
• Symptom might be intermittent
  - E.g., timing problems, or hardware interaction
• May be hard to reproduce the erroneous run
• Requires unusual skills
  - Combines brain teasers with the annoying recognition that you made a mistake
  - Large variance in debugging skills

Debugging Strategies

• Get the failed program to fail repeatably, on the simplest input
  - Add that input to your regression suite
• Brute force
  - Add print statements and read the output
• Backtracking
  - Work backwards from where symptom appears
• Cause elimination
  - Hypothesize causes, then create tests to validate
**GDB**

- The Gnu DeBugger
- A very real debugger
  - Widely used
  - Runs on everything
- Also, a classic implementation
  - Mostly standard debugger technology
  - Similar debuggers exist for many languages

**Design Decisions**

- Works with object code
  - Runs object code, instruments object code, etc.
  - Works for many source languages
  - Can be used in absence of source code
- Issues
  - Must map accurately between source/object code
  - Must deal with many different machines
  - Must be well-integrated with the compiler

**Debugger Architecture**

- Three major pieces:
  - User interface
  - Symbol piece
    - Mapping from source to object code
  - Execution piece
    - Manipulating, running object code

**Concepts**

- Debuggers use compiler terminology
  - Some cs164 background helpful here
- Symbols
  - Variable names, procedure names
- Source code
  - The program you write
- Object code
  - The compiled program

**User Interface**

- Not much to say, except that it’s classic Unix

  necula% gdb
  (gdb) file foo
  Reading symbols from foo … done
  (gdb) break main
  Breakpoint 1 at 0x40107e: file foo.c line 10
  (gdb) run
  Starting program: /home/necula/foo.exe
  Breakpoint 1, main() at foo.c:10
  10      printf("Hello world!")

**Object Code**

- Machine code
  - E.g., 0x45, 0x28, 0x51, 0x40
- Can be disassembled
  - E.g., call 0x405128
- Additional stuff
  - Flags (size of data, stack, address of entry point)
  - Relocation tables (load code at different addr.)
  - Symbols (map code and data addresses to names)
  - Debugging info (map code addr. to line numbers)
- Useful tool: objdump
Symbol Piece

- Insight: The compiler knows all this
- Solution: Dump the compile-time information into extra tables in the object code
  - At least when debugging is on
- Source-line information is not completely accurate when optimizations are on
  - The compiler loses track of position

Execution Piece

- Run object code
- Disassemble object code
- Manipulate stack frames
- Set breakpoints

Features

- Breakpoints
- Single stepping
- Host/Target

Breakpoints

- The fundamental debugging primitive
- How does it work?
  - Via an object code rewriting hack
  - To stop at line 10, write an invalid opcode at line 10
  - Trap resulting fault, recover, and switch to the UI
- Invalid opcode should be as small as possible

Single Stepping

- To single step:
  - Set breakpoint at next instruction
  - Resume execution
  - Trap exception, clear breakpoint, repeat
- Or
  - Use software interpreter
  - Interpret instructions to next source statement

Other Features

- Other features based on breakpoints
  - Skip over function call
  - Put breakpoint at the return site
  - Break on nth execution of a statement
  - Break when an expression becomes true
  - Break when an expression changes value
- Or, inspect execution state
  - Print the call stack
  - Inspect data values
  - Etc.
Host/Target

- Gdb can be used to debug a program on a remote machine
  - Gdb runs on the host
  - Program runs on the target
- Introduces cross-architecture issues
- What is the application for this feature?

Multithreading

- Debugging multithreaded code is hard
  - Why?
- Use the ability to attach to a process
  - Interrupt a running process
  - Put it under debugger control
  - Then set breakpoints, etc.

Mulithreaded Code Debugging Hack

Add the following code to each process

```c
Die() {
    printf("Failing, process id is \%d", getpid());
    volatile int waiting = 1;
    while (waiting) { sleep(1) ;
    ...
}
Call here on assertion failure
```

Print pid on console

Program waits here for you to attach

In debugger, set waiting to 0 to release program from loop, set breakpoint after loop

Just-in-Time Debugging

- Start the debugger when the program fails an assertion
  - You do not have to reproduce the run
  - Microsoft Visual Studio does this
- Can simulate this with gdb's ability to attach to a process
  - On assertion failure, you trigger an external process that starts gdb

Opinions: Debugger Drawbacks

- Tight integration of compiler and debugger
  - Wide interface
  - Does not scale well with compiler complexity
- Handling object file formats a big deal
  - Engineering galore
  - Another wide interface

A Big Problem with Debuggers

- Seemingly unavoidable lack of support for optimized code
- Makes it difficult to debug "the real thing"
  - Find compiler bugs
  - Find timing-dependent bugs
  - Find resource/performance bugs
- True for any known approach to debuggers
- A lot of deployed code has optimizations off
Opinions: Advantages

- Works even if source is not available
  - Albeit crippled
- Responsive
  - Interactive experience is good
  - Scales well with object code size

Research Topic: Time-Travel Debuggers

- When debugging, often want to go back in time
  - Find out what happened just before a crash
  - Work backwards towards the cause
- Idea: Build a debugger that can replay computations

Time-Travel

- Essentially, checkpoint/replay
- Save checkpoints during computation
  - That is, save entire state of computation
- To travel to time t
  - Return to last checkpoint before time t
  - Rerun computation up to time t

Issues

- How many checkpoints?
  - Tradeoff between space usage and query time
  - LRU-based policy is natural
    - We are likely to revisit a recently visited time
- I/O is a problem
  - Must log and replay I/O events
  - Exposed to however much I/O the program wants to do

Benefits

- Time travel makes debugger internals easier
  - Need not set precise breakpoints
  - Can always overshoot and then time travel backwards
- Gives the user a new tool
  - E.g., travel backwards to the time when some property first became true

Costs

- A replay debugger is not cheap
  - Factor of 3 in speed
  - Factor of 5 in memory
- And this is for declarative languages
  - Discourages updates to program state
  - A style that makes checkpointing cheaper
Debugging Conclusions

- Debugging is hard

- Can increase dramatically effectiveness with the right tools
  - Debuggers
  - Time-travel
  - Just-in-time

- Or automated debugging tools (next time)