Debuggers

CS195
Lecture 5

Outline

• How do interactive debuggers work?
  – A “real” debugger (gdb)
  – A research debugger

GDB

• A very real debugger
  – Widely used
  – Runs on everything

• Also, a classic implementation
  – Mostly standard debugger technology

Concepts

• Debuggers use compiler terminology
  – Some cs143 background helpful here

• Symbols
  – Variable names, procedure names

• Source code
  – The program you write

• Object code
  – The compiled program

Design Decisions

• Works with object code
  – Runs object code, instruments object code, etc.

• Issues
  – Must map accurately between source/object code
  – Must deal with many different machines
  – Must be well-integrated with the compiler

Architecture

• Three major pieces:

  • User interface
    – Really interfaces

  • Symbol piece
    – Mapping from source to object code

  • Execution piece
    – Manipulating, running object code
User Interface

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

Symbol Piece

- Need to match source code constructs with object code constructs
- Need to evaluate expressions
- Both require knowledge of
  - Variables
  - Types
  - Functions
  - Line numbers

Symbol Piece (Cont.)

- Insight: The compiler knows all of this
- Solution: Dump the compile-time information into extra tables in the object code
  - At least when debugging is on
- Implies we must disable most optimization
  - Otherwise, even the compiler loses track of the position of source lines
  - Note: gdb works with optimization, it is just not nearly as helpful with optimization

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 hardware interpreter
  - Interpret instructions to next source statement

**Other Features**

- Other features based on breakpoints
  - Skip over function call
  - Break on nth execution of a statement

- Or exploit compile-time symbol information
  - Print the call stack
  - 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

- Use the ability to attach to a process
  - Interrupt a running process
  - Put it under debugger control
  - Then set breakpoints, etc.

**Multipthreaded Code Debugging Hack**

Add the following code to each process:

```
Die() {
    printf("Failing, process id is \%d", getpid());
    volatile int waiting = 1;
    while (waiting) { sleep(1) };
    print pid on console

    Program waits here for you to attach

    Call here on assertion failure
   
    In debugger, set waiting to 0 to release program from loop, set break point after loop.
}
```

**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

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
  - Redo a computation with different values
  - Find out what happened just before a crash
- 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
  - Factor of 5 in code size

• And this is for functional languages
  - Discourages updates to program state
  - A style that makes checkpointing cheaper