CSCI 414 / 553: Networking III
Unix Network Programming
Spring 2008

Debugging

Today's Agenda

- A word about Lab 01 & tasks for class.
- Lecture on Debugging
- Lab 02 (@ 10:15 till 10:45)

Introduction

- You're going to spend half your professional life debugging
  - So you should learn how to do it systematically
- Talk about tools first
  - They'll make everything else less painful
- Then some techniques
- Assume for now that you built the right thing the wrong way
  - Requirements errors are actually a major cause of software project failure
  - But out of scope for this course

Whats Wrong with Print Statements

- Many people still debug by adding print statements to their programs
- It's error-prone
  - Adding print statements is a good way to add typos
  - Particularly when you have to modify the block structure of your program
- And time-consuming
  - All that typing…
  - And (if you're using Java, C++, or Fortran) all that recompiling…
- And can be misleading
  - Moves things around in memory, changes execution timing, etc.
  - Common for bugs to hide when print statements are added, and reappear when they're removed

Symbolic Debuggers

- A debugger is a program that runs another program on your behalf
  - Sometimes called a *symbolic* debugger because it shows you the source code you wrote, rather than raw machine code
- While the target program (or debugee) is running, the debugger can:
  - Pause, resume, or restart the target
  - Display or change values
  - Watch for calls to particular functions, changes to particular variables, etc.
- Do *not* need to modify the source of the target program!
  - Depending on your language, you may need to compile it with different flags
- And yes, the debugger modifies the target's layout in memory, and execution speed…
  - ...but a lot less than print statements…
  - ...with a lot less effort from you

Debugger Features

- Interactive debuggers typically show:
  - The source code
  - The call stack
  - The values of variables that are currently in scope
    - I.e., global variables, parameters to the current function call, and local variables in that function
  - A panel displaying what your program has printed to standard output and/or standard error
- We'll use gdb & kdbg gui in this lecture

Kinds of Debuggers

- There may be several ways to get into the debugger
  - Launch the debugger, load the target program, and start work
  - Run the debugger with the target program as a command-line argument
  - Switch into debugging mode in the middle of an interactive session
- Sometimes also do post mortem debugging
  - When a program fails badly, it creates a core dump
    - Copies all of its internal state to a file on disk
  - Load that dump into the debugger, and see where the program was when it terminated
    - Not as good as watching it run…
    - …but sometimes the best you can do

Integrated Development Environments

- Debuggers are usually part of Integrated Development Environments (IDEs)
  - These usually contain many other tools as well, including:
  - A class browser that presents an outline of the project's modules, classes, functions, variables, etc.
  - A code assistant that presents context-sensitive help and documentation
- Tools like this are available for every modern language
  - Microsoft Visual Studio on Windows
  - Eclipse

Command Line Debuggers

- Many of today's debuggers are GUIs wrapped around older command-line debuggers
- Most widely used of these is GDB
  - Supports many languages, on many platforms
  - But no one ever said it was easy to learn

Inspecting Values

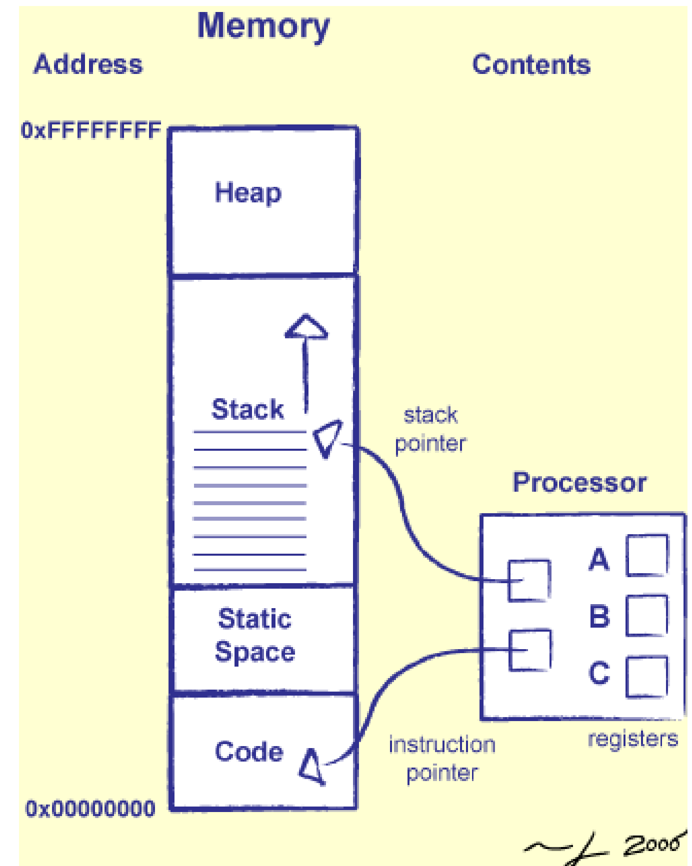
- Use the debugger to set breakpoints in the target program
  - Tells the target program to pause when it reaches particular lines of code
- When the target program is paused, the debugger can display the contents of its memory
- Most debuggers can also evaluate expressions using the current values of variables
  - E.g., type in 2*x<0, debugger displays False

Controlling Execution

- The debugger can also:
  - Single-step (i.e., execute one statement at a time)
  - Step into function calls
  - Step over them, or
  - Run to the end of the current function
- Allows you to see:
  - How values are changing
  - Which branches the program is actually taking
  - Which functions are actually being called
- Debuggers really should be called “inspectors”

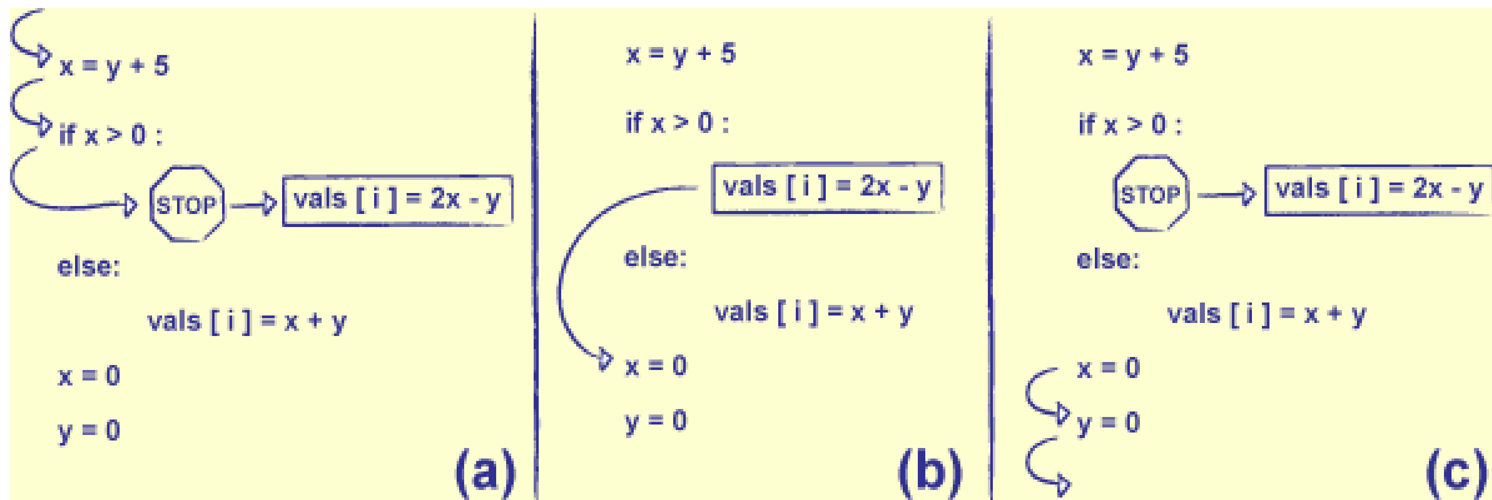
Under the Hood

- Programs are “just” data
- Some bytes hold values that represent instructions
  - Each statement in the source program typically corresponds to several instructions
  - The compiler (or interpreter) keeps track of which lines of code produced which instructions
- Other bytes hold constants and variables
  - The static space contains constant strings, magic numbers, etc.
  - The call stack holds function parameters
  - The heap is all dynamically-created objects
- Some registers
  - An instruction pointer that keeps track of what to execute next
  - A stack pointer to keep track of the call stack
  - Miscellaneous other registers for doing arithmetic, etc.



Implementing Breakpoints

- To set a breakpoint on a particular line, the debugger:
  - Figures out which instructions were produced for the statement on that line
  - Copies the first of those instructions to a safe place
  - Replaces it with a HALT instruction



Implementing Breakpoints

- When the target program reaches the HALT instruction, it signals the debugger
  - Which can then inspect the target program's memory
- When the user wants the program to resume, the debugger:
  - Puts the instruction at the breakpoint location back in place
  - Tells the program to execute that instruction
  - Replaces the instruction with a HALT once again
  - Tells the program to run normally

Inspecting More Values

- Debugger lets you move up and down the call stack
  - Allows you to run to the problem, then figure out how you got there
- Also lets you modify values
  - If you have a theory about why a bug is occurring:
    - Run to that point
    - Set variables' values (e.g., set max_temp to -1)
    - Resume execution
  - Sometimes used to test error handling code
    - Easier to change time_spent_waiting to 600 seconds in debugger than to pull out the network cable and wait…

Conditional Breakpoints and Watchpoints

- The program only stops at a conditional breakpoint if some condition is met
  - E.g., loop index greater than 100, or filename argument is None
  - Much more efficient than single-stepping from the start of the program
- Some debuggers also support watchpoints
  - Have the debugger watch every write to memory
  - Halt when anything, anywhere, modifies a particular variable
  - Slow the program down a lot
    - Typically a factor of 100 or more
  - But sometimes the only practical way to find out when a particular list value is being overwritten

Logging

- Sometimes printing is the right thing to do
  - Collecting information for later analysis (e.g., web server
  - Not expecting anything to go wrong, but want to be able to trace execution leading up to fault if one does occur
- Many systems use logging to record information in a structured, manageable way
  - Separate different levels of information
    - Debugging vs. warning vs. critical
  - Separate information about different things
    - Login/logout vs. backup
  - Send information to different destinations
    - Files vs. database vs. sys admin's pager

Logging Levels

- Every system is different, but the following are fairly standard
- DEBUG: only want to see it when debugging a problem
  - Be careful not to leave anything in a released product that you don't want customers to be able to turn on
- INFO: information about normal operations
  - The sort of thing that goes into a web server log
- WARNING: something that a human being should pay attention to
  - E.g., failed login attempt, or site not found
- ERROR: something has gone wrong inside the software
- CRITICAL: something has gone very wrong inside the software
  - System is about to crash, reactor is about to melt down, etc.

Agan's Rule

- Many people make debugging harder than it needs to be by:
  - Using inadequate tools
  - Not going about it systematically
  - Becoming impatient
- Agans' Rules [Agans 2002] describe how to apply the scientific method to debugging
  - Observe a failure
  - Invent a hypothesis explaining the cause
  - Test the hypothesis by running an experiment (i.e., a test)
  - Repeat until the bug has been found

Rule 0: Get it Right the First Time

- The simplest bugs to fix are the ones that don't exist
- Design, reflect, discuss, then code
  - "A week of hard work can sometimes save you an hour of thought."
- Design and build your code with testing and debugging in mind
  - Minimize the amount of "spooky action at a distance"
  - Minimize the number of things programmers have to keep track of at any one time
  - Train yourself to do things right, so that you'll code well even when you're tired, stressed, and facing a deadline
- "Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?" (Brian Kernighan)

Rule 1: What Is It Supposed to Do?

- First step is knowing what the problem is
  - “It doesn't work” isn't good enough
  - What exactly is going wrong?
  - How do you know?
  - You will learn a lot by following execution in a debugger and trying to anticipate what the program is going to do next
- Requires you to know how the software is supposed to behave
  - Is this case covered by the specification?
  - If not:
    - Do you have enough knowledge to extrapolate?
    - Do you have the right to do so?
- Try not to let what you want to see influence what you actually observe
  - It's harder than you'd think [Hock 2004]

Rule 2: Is It Plugged In?

- Are you actually exercising the problem that you think you are?
  - Are you giving it the right test data?
  - Is it configured the way you think it is?
  - Is it the version you think it is?
  - Has the feature actually been implemented yet?
  - Why are you sure?
    - Maybe the reason you can't isolate the problem is that it's not there
- Another argument in favor of automatic regression tests
  - Guaranteed to rerun the test the same way each time
- Also a good argument against automatic regression tests
  - If the test is wrong, it will generate the same misleading result each time

- You can only debug things when they go wrong
- So find a test case that makes the code fail every time
  - Then try to find a simpler one
  - Or start with a trivially simple test case that passes, then add complexity until it fails
- Each experiment becomes a test case
  - So that you can re-run all of them with a single command
  - How else are you going to know that the bug has actually been fixed?
- Use the scientific method
  - Formulate a hypothesis, make a prediction, conduct an experiment, repeat
  - Remember, it's computer science, not computer flip-a-coin

Alternatives

- What if you can't make it fail reliably?
  - Problem involves timing, network load, etc.
  - Or you just don't know enough about the cause yet
- Use post-mortem inspection
  - But then you have to reason backwards to figure out why the program crashed
- Or logging
  - But this can distort the program's behavior
  - And you'll have to wade through a lot of irrelevant information

Rule 4: Divide and Conquer

- The smaller the gap between cause and effect, the easier the relationship is to see
- So once you have a test that makes the system fail, use it isolate the faulty subsystem
  - Examine the input of the code that's failing
  - If that's wrong, look at the preceding code's input, and so on
- Use assert to check things that ought to be right
  - “Fail early, fail often”
  - A good way to stop yourself from introducing new bugs as you fix old ones
- When you do fix the bug, see whether you can add assertions to prevent it reappearing
  - If you made the mistake once, odds are that you, or someone, will make it again
- Another argument against duplicated code
  - Few things are as frustrating as fixing a bug, only to have it crop up again elsewhere

Rule 5: Change One Thing at a Time, For a Reason

- Replacing random chunks of code unlikely to do much good
  - If you got it wrong the first time, what makes you think you'll get it right the second? Or the ninth?
  - So always have a hypothesis before making a change
- Every time you make a change, re-run all of your tests immediately
  - The more things you change at once, the harder it is to know what's responsible for what
  - And the harder it is to keep track of what you've done, and what effect it had
  - Changes can also often uncover (or introduce) new bugs

Rule 6: Write It Down

- Science works because scientists keep records
  - "Did left followed by right with an odd number of lines cause the crash? Or was it right followed by left? Or was I using an even number of lines?"
- Records particularly useful when getting help
  - People are more likely to listen when you can explain clearly what you did

Rule 7: Be Humble

- If you can't find it in 15 minutes, ask for help
  - Just explaining the problem aloud is often enough
  - "Never debug standing up." (Gerald Weinberg)
- Don't keep telling yourself why it should work: if it doesn't, it doesn't
  - Never debug while grinding your teeth, either…
- Keep track of your mistakes
  - Just as runners keep track of their time for the 100 meter sprint
  - "You cannot manage what you cannot measure." (Bill Hewlett)
- And read [Zeller 2006] to learn more

Summary

- Debugging is *not* a black art
- Like medical diagnosis, it's a skill that can be studied and improved
- You're going to spend a lot of time doing it: you might as well learn how to do it well