Lecture 24: Debugging and *gdb*

Kenneth M. Anderson

Software Methods and Tools

CSCI 3308 - Fall Semester, 2003

Today's Lecture

- Debugging
  - The concept
  - Terminology
  - Types of Debuggers
- gdb - The GNU Debugger

Debugging

- Debugging is the process of finding faults in a program after a failure has been detected
- Typically
  - a program fails a test case (or a bug is reported)
  - a programmer is given the test case, checks out the associated source code, and "debugs" the program until the fault has been corrected
  - the program, having been fixed, passes the test case and no longer exhibits the failure

Finding Faults

- The hard part of debugging is locating faults
  - Once a fault is located, it is typically straightforward to fix
- Having a failing test case is helpful
  - because the fault must be in some part of the code that was executed by the program before the failure occurred
  - The entire program does not have to be examined

Bridging the Gap

- One problem with faults is that they are not necessarily located "near" their associated failure
- Therefore, "bring the failure close to the fault"
  - the idea being that if we do, less code needs to be examined to find the fault
  - Thus, we often insert "print" statements into code to force the failure to appear as soon as possible after the fault
  - Taken to the extreme, a programmer should be able to see the value of any variable at any time during a program's execution

Debugging Tools

- This is the purpose of debugging tools
  - called "debuggers"
- A debugger allows a programmer to monitor the internal state of a program while its executing
- Two types of debuggers
  - Interpretive
  - Direct Execution

Types of Debuggers

- Interpretive debugger
  - works by reading a program and simulating its execution one line at a time
- Direct Execution Debugger
  - works by running the actual program in a special mode where the debugger can read and write the program's memory

Two styles of use

- Line-at-a-time
  - A programmer loads a program into the debugger and "steps" through the program one line at a time.
  - The debugger stops the program after each line and gives the programmer a chance to check the program's variables to see if it is operating correctly
- Breakpoints
  - A programmer loads a program into the debugger and specifies "breakpoints" at various locations in the program
  - The program runs until it hits a breakpoint

More on Breakpoints

- How are breakpoints useful?
  - If you think you have a function that might have a fault
    - set a breakpoint at the beginning of the function
    - set another breakpoint at the end of the function
    - if a program's data is correct at the beginning of the function but incorrect at the end, then there is a fault in the function
  - They also allow you to skip over "init" code and debugged code quickly; letting the programmer focus on finding the fault

Implementing Breakpoints

- How do debuggers implement breakpoints?
- Interpretive Debuggers essentially execute as a while loop

```
while (…)
    Read the next line of the program
    Is there a breakpoint on this line?
        Yes, stop and print a prompt
    Execute this instruction
end while
```

- Direct Execution debuggers implement breakpoints by "cutting-and-pasting" the executable instructions of the program itself
  - that is, they insert instructions into the program that notifies the debugger when a breakpoint has been hit

Editing Variables

- Debuggers let the programmer explore "what-if" scenarios
  - You can execute a program to a certain point, and then alter the values of the program's variables
  - You can thus explore unusual cases in the code
  - Plus, if a bug occurs, you can correct an incorrect value and see how far the program goes before it encounters another fault

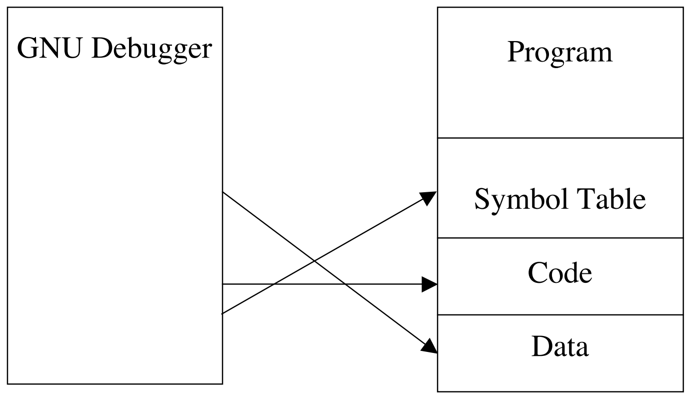
Advantages

- Advantages of interpretive debuggers
  - Easier to program the debugger
  - Safer, a program cannot crash the machine (just the debugger)
- Advantages of direct execution debuggers
  - Faster
  - More accurate, the actual program instructions are being executed

The GNU Debugger

- The GNU Debugger (gdb) is a direct execution debugger
  - There are separate processes for the debugger and the program
  - All of the variables are kept in the program's process
  - GDB can read/write the program's memory

GDB Architecture



Preparing to use the Debugger

- In order for gdb to process a program, it must be compiled using the "-g" flag

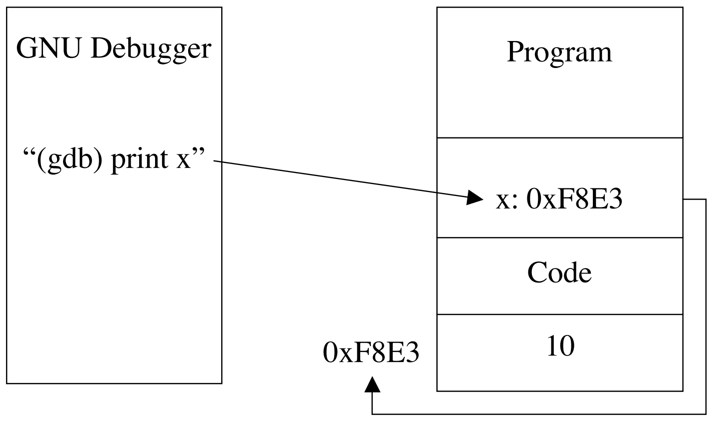
```
gcc -c -g program.c
gcc program.o -o program
```

- This flag tells the compiler to include a symbol table into the compiled program

What's a symbol table?

- A symbol table records the mapping between a program's variables and their locations in memory
  - Executable code uses machine addresses to reference memory. The variable "x" might be stored at address 0xF8E3
  - So rather than typing "print 0xF8E3" to see the value of the variable "x", you can instead type "print x" and gdb uses the symbol table to locate and print the correct memory location

An Example



More on the Symbol Table

- A symbol table is not automatically included in a program because
  - it takes up space
  - its not required for a program to execute
- You can still use gdb on a program that doesn't have a symbol table
  - however, you then have to type things like "print 0xF8E3" since the debugger will not be able to map these values to variable names

Debugging Programs

- A debugger does not automatically execute the program to be debugged
  - You need to have a chance to configure a program's values, set breakpoints, etc. before running it
- Instead, it loads the program into a separate process and prints a prompt

```
gdb program
GNU gdb 4.17…
(gdb)
```

Running programs in GDB

- Typing "run" at the prompt will run the program as normal
  - e.g. it will function as if you had invoked it from the shell
- In order to debug the program, you need to set a breakpoint
  - (gdb) break main
  - Breakpoint 1 at 0x229c
  - (gdb) run
  - Starting program: program
  - Breakpoint 1, 0x229c in main ()
  - (gdb)

Supplying Input to a Program

- Normally, you use shell redirection to supply input to a program
  - %program < test-input
- If you want to do this to a program that you want to debug, you may try
  - %gdb program < test-input
- Unfortunately, this supplies the input to gdb not to the program! So, you need to do this:
  - %gdb program
  - (gdb) run < test-input

Program Crashed. Core Dumped

- When a program crashes in Unix, it creates a file called “core”
  - Core stands for “Core Memory Dump”
- The entire contents of a program’s memory is dumped into this file
- gdb can read this file to tell you
  - what instruction was being executed when the crash occurred
  - what the value of the program’s variables were at the time of the crash

To Read a Core File

- An example

```
%program
Segmentation fault (core dumped)
%gdb program core
```

- When debugging a program using its core file, you cannot continue execution
  - because the program crashed!
  - but you can “take a look around”

More on Debugging

- When gdb hits a breakpoint, it is possible to look at the program’s “call stack”
  - When a function is called, it is placed on the call stack
  - When a function returns, it is popped off the call stack and control returns to the calling function
- gdb provides operations for examining this stack

Example

```
main() {
      int i;
      i = 2;
      func(4);
}

func (int i) {
      …
}
```

- (gdb) backtrace
- #0 func (i=4) at main.c:8
- #1 0x22b0 in main() at main.c:4
- (gdb) print i
- 4
- (gdb) frame 1
- #1 0x22b0 in main () at main.c:4
- (gdb) print i
- 2

Common Commands

- run - execute program
- b <number> or <name> - break at line number or subroutine
- list - list source code
  - list filename:number
- step - execute one line (step into subs)
- continue - continue execution until next breakpoint or end of program
- next - execute next instruction (step over subs)
- bt - view call stack
- frame - select frame on call stack
- print <variable> or <expression>