CSE 303 Concepts and Tools for Software Development

Richard C. Davis
UW CSE – 11/22/2006

Lecture 20 – Debuggers

Administravia

- HW6 Due 30 Minutes Ago!
- Societal Implications 4 next Wednesday
  - We'll resume discussion of DRM
  - No new reading
- Short Paper Instructions are posted
  - Due last day of class
  - This is 2 days after HW7
- HW7 Posted by Monday

Where We Are

- Tools for working on larger projects
  - Specification/Testing
  - Version Control Systems
  - Build Scripting

Today

- Debuggers
  - Role
  - How they work
  - The art of debugging
- GDB

Role of a Debugger

- Helps you see what's going on in program
- More accurate name: "Execution Monitor"
- Debugger Capabilities
  - Start program with arguments
  - Suspend execution
    - At predefined "breakpoints"
    - Possible to break on some condition
  - Examine suspended state of program
  - Change the values of variables (sometimes)

How Debuggers Work

- Program has special links to source code
  - These make programs much larger
    - Most projects have "Debug" and "Release" builds
    - External libraries may also have debug versions
  - `gcc` & `g++` add debug info with `-g` flag
- OS hooks for examining program state

The Art of Debugging

- A debugger won't solve all your problems
  - Stopping program too late to find problem
  - Trying to "debug" the wrong algorithm
  - "Debugging" vs. "Understanding the program"
- Debugging C/C++ vs. Java
  - No crashes != Correct program
  - Java easier to debug: No crashes or memory errors
  - But programming Java is "easier" for same reason

Debugging with GDB

- Running gdb
  - Command Line: `gdb programname`
    - Source files should be in same directory
  - Emacs: M-x gdb
    - Current file must be in directory of program/source
- Note: There are many other debuggers
  - `dbx`, `jdb` (for Java)
  - Debuggers integrated into IDEs
    - But concepts are the same

Example: Locating a Crash

- Approach 1: Execute program in gdb

```
-bash-3.1$ gdb bug4
…
(gdb) run
…
Program received signal SIGABRT, Aborted.
0xffffe410 in __kernel_vsyscall ()
(gdb) where
```


Example: Locating a Crash

```
(gdb) where
#0  0xffffe410 in __kernel_vsyscall ()
#1  0x4320bee9 in raise () from /lib/libc.so.6
#2  0x4320d4f1 in abort () from /lib/libc.so.6
#3  0x4324053b in __libc_message () from
  /lib/libc.so.6
#4  0x43247a68 in _int_free () from /lib/libc.so.6
#5  0x4324af6f in free () from /lib/libc.so.6
#6  0x43dc96c1 in operator delete () from
  /usr/lib/libstdc++.so.6
#7  0x080485d7 in ~A (this=0xbfbdb744) at bug4.cpp:5
#8  0x08048567 in main () at bug4.cpp:19
(gdb) up
(gdb) <pressing return repeats previous command>
```


Example: Locating a Crash

- Approach 2: Examine a core file
  - Need to set max size allowed for core files
    `ulimit -c 16000`
  - Run program as usual
    `Aborted (core dumped)`
  - Examine core file with gdb
    `dgb bug4 core.10050`
    `(gdb) where`
  - Same output as Approach 1

Example: Suspending a Program

- Approach 1: Send interrupt

```
gdb heaptest
run 10          ←Note here that you can supply arguments
(user presses Ctrl-c, or Ctrl-c Ctrl-c if in Emacs)
Program received signal SIGINT, Interrupt.
0xffffe410 in __kernel_vsyscall ()
(gdb)
```

Example: Suspending a Program

- Approach 2: Place a breakpoint

```
gdb heaptest
break heaptest.c:26
run 10
Program received signal SIGINT, Interrupt.
0xffffe410 in __kernel_vsyscall ()
(gdb)
```


Working with Breakpoints

- Function Names
  `break main`
- Within files
  `break heap.c:HeapInit`
- Within methods
  `break Point::GetX`
- Delete all breakpoints
  `delete`
- Clear one breakpoint
  `clear heap.c:HeapInit`
- Conditional breakpoint
  `break Point::SetX if newX==1`

Inspecting the Program

- Inspecting arguments and local variables

```
(gdb) info args          ←Show arguments
(gdb) info locals        ←Show local variables
(gdb) info variables     ←Show locals and globals
(gdb) p variable_name    ←Print a variable
```


Inspecting the Program

- Where are we?

```
(gdb) where      ←Show call stack
(gdb) frame      ←Show current activation record
(gdb) up         ←Move "up" the call stack
(gdb) down       ← Move "down" the call stack
(gdb) l          ←Print 10 lines of context
```

- Names of variables depend on current stack frame

Other Commands

- Executing Step-by-step

```
(gdb) n    ←Execute one statement and stop at next
(gdb) s    ←Step inside function
(gdb) c    ←Continue until next breakpoint
```

- Quitting

```
(gdb) quit
```


Summary

- Debuggers important for fast development
- Understand what the tool provides you
- Use it to accomplish specific tasks
  - "I want to know the call stack when I get the NULL-Pointer dereference"
- Avoid command line
  - Use Emacs
  - Use an IDE

Reading

- Programming in C
  - Chapter 18: Debugging Programs

Next Time

- Profilers