How To Debug Programs, Part 2
Debugging Tools

CS24 – Spring 2012

CS24 and Debugging (2)

- Last time, covered a more general question:
  - Part 1: What are the basic principles and approaches of debugging?
  - [Mostly] independent of specific language, platform or toolset
- This time:
  - Part 2: What tools and approaches can I use to debug my C and IA32 assembly language programs in CS24?
- Tonight: Cover GDB, DDD, and Valgrind tools

Last Time: Reproducing Failures

- Last time, discussed importance of reproducing failures:
  1. **So you can watch it fail.** You can see exactly what the program was doing as it crashed and burned.
  2. **So you can zero in on the cause.** If the program fails in some circumstances but not in others, this will give you *hints* as to what part of the program actually contains the defect.
  3. **So you can test if you actually fixed it.** If you can reliably cause the failure, and your fix makes it go away, you win!
- Debugging tools help you watch your program execute in a less intrusive way
  - Alternative is adding debug-printout statements…
  - Not always possible – e.g. if you can't change the binary, or if program is in assembly language, etc.

GDB: The GNU Debugger

- gdb is the GNU Debugger
  - Designed to play very well with gcc, g++, as, etc.
- Understands different source languages you might use
  - Properly displays C code, C++ code, IA32 assembly code…
  - Understands many expressions written in these languages too
- Idea: run your program in the debugger
  - Invoke gdb with your binary file…
    - gdb starts up and loads your program into memory
  - Possibly perform initial setup for debugging your program
    - e.g. set breakpoints at known trouble-spots
  - Tell the debugger to run your program!
    - Interact with the debugger to execute instructions step by step, watch how state is manipulated by your program, etc.

Setting Up For Debugging

- Normally, the compiler doesn't record variable names, function names, source code, etc. in the binary file
  - Names are translated into numeric addresses
  - C statements are translated into (possibly many) IA32 machine code instructions
  - Don't want to store information about the sources, since it takes up space, makes the program slower to load, etc.
- But, gdb needs this info to facilitate easier debugging
- Must tell compiler to leave this info in the resulting binary
  - Can pass –g flag to gcc (and to as)
  - Causes compiler to include debugging symbols in the binary file
  - Then, gdb can load this info when it loads your program

Setting Up For Debugging (2)

- Should also tell the compiler to turn off optimizations
  - Compiler may reorder machine-code operations to optimize execution of the program
    - (applies rules to generate an equivalent program – produces same results, but may not be in exactly the same order you specified)
  - Can easily make debugging very confusing!
  - Specify –O0 to gcc (as doesn't optimize anything)
- Can set this up in the Makefile:
  - At top of Makefile, put these variable definitions:

    CFLAGS = –g –O0

    ASFLAGS = –g
  - (Read about implicit build rules in the Make Primer to understand how these variables are used in compilation.)

Setting Up For Debugging (3)

- Don't forget to recompile everything when setting up to debug!
  - Every object-file needs to be regenerated to include debug info
  - Sometimes students forget to recompile after changing the makefile to include debugging symbols
    - make clean
    - make
  - Should see "–g –O0" in the commands issued by make
  - If not, your edits may not be correct
- Most of the CS24 makefiles include the –g flag
  - (We kinda expect you'll be needing to debug your stuff…)
- Once you have included debug info, ready to use gdb!

Example: "My program crashes!"

- A common experience:

  [*user@host*]> ./nummain

  Segmentation fault [On MacOSX: "Bus error"]

  [*user@host*]>

  - What is going on?!? Where is my program crashing?!
- gdb makes it easy to find where your program is crashing

  [user@host]> gdb nummain

  …gdb startup information…

  (gdb)

  - Next, start your program running within the debugger

    (gdb) run [*starts the program from main()*]

    Starting program: nummain
  - When your program crashes, gdb will come back and tell you

Example: "My program crashes!" (2)

- When your program crashes:

```
(gdb) run
Starting program: nummain

Program received signal SIGSEGV, Segmentation fault.
0x08048546 in add_num (lst=0xbffff3c4, num=3) at numlist.c:16
16      lst->tail->next = n;
```

- Notice: gdb tells you *exactly* where the crash occurred!
  - What function, what the arguments to the function were, and what source file the function is in
  - Additionally, shows the exact line that had the problem!

Example: "My program crashes!" (3)

- Can look at the surrounding code in context:

```
(gdb) list                                  [shows 10 lines of code]
11 void add_num(list *lst, int num) {
12     node *n = (node *) malloc(sizeof(node));
13     n->value = num;
14     n->next = NULL;
15
16     lst->tail->next = n;
17
18     lst->tail = n;
19     lst->size++;
20 }
```

  - Can see the code in context
  - "Oh, I forgot to check if the list was initially empty…"

Example: "My program crashes!" (4)

- Can also look at variables in the program:

```
(gdb) print lst                          [Hmm, lst looks okay…]
$1 = (list *) 0xbffff3c4
(gdb) print *lst
$2 = {head = 0x0, tail = 0x0, size = 0}
(gdb) print lst->tail                    [Ah ha!  lst->tail is NULL.]
$3 = (node *) 0x0
(gdb) print n
$4 = (node *) 0x804b008
(gdb) print *n                           [Well, at least n is okay…]
$5 = {value = 3, next = 0x0}
```

- Note that gdb understands C syntax!
  - Can even do things like: print *(lst->head->next->next)

Example: "My program crashes!" (4)

- Can see the exact call-sequence for your program:

  (gdb) where [*shows the entire call-stack*]

  #0 0x08048546 in add_num (lst=0xbffff3c4, num=3) at numlist.c:16

  #1 0x08048651 in main () at nummain.c:10

  - (this program isn't very exciting)

GDB Breakpoints

- Can also set breakpoints in your program
  - When gdb reaches the breakpoint, it will stop automatically
  - Can only set breakpoints while the program is stopped!
  - (Frequently must set these up before running the program.)
- Fixed previous bug, but still have odd behavior:
  - [*user@host*]> ./nummain
  - Original list: 3 1 4 1 5
  - Reversed list: 5 1 4 1 3
  - Original list: 5 9 2 6 5 4
  - Reversed list: 4 5 6 2 9 5
  - Cleaning up list.
  - Hmm, losing some of our digits in the last part of the program
  - Maybe reverse() has a bug in it…

GDB Breakpoints (2)

- Set breakpoint in reverse():

```
gdb nummain
…gdb startup information…
(gdb) break reverse
Breakpoint 1 at 0x80485eb: file numlist.c, line 50.
(gdb)
```

- Every time gdb reaches the breakpoint, it will stop executing and let you poke around

```
(gdb) run
Starting program: nummain
Original list:  3 1 4 1 5

Breakpoint 1, reverse (lst=0xbffff3c4) at numlist.c:50
50      prev = NULL;
(gdb)
```

GDB Breakpoints (3)

- As before, can list the code, or print various values
- Can also single-step through the code:
  - step — Steps to next line of code (steps into function calls)
  - next — Steps to next line of code (steps over function calls)
  - continue — Resumes running program, until next breakpoint, or whenever the program terminates.
- Example, continued:

```
Breakpoint 1, reverse (lst=0xbffff3c4) at numlist.c:50
50      prev = NULL;
(gdb) next
51      curr = lst->head;
(gdb) next
53      while (curr != NULL) {
(gdb) next
54          next = curr->next;
```

GDB Breakpoints (4)

- Can set breakpoints in several ways:
  - break *label*
    - Can specify a function name, or a label in your IA32 assembly code
  - break *filename:line*
    - Can specify a filename and line number as well
- Every breakpoint is given a numeric ID

```
(gdb) break reverse
Breakpoint 1 at 0x80485eb: file numlist.c, line 50.
(gdb)
```

- Can enable and disable breakpoints as well:

```
(gdb) disable 1          [disabled reverse() breakpoint]
(gdb) enable 1           [re-enables reverse() breakpoint]
```

GDB Commands

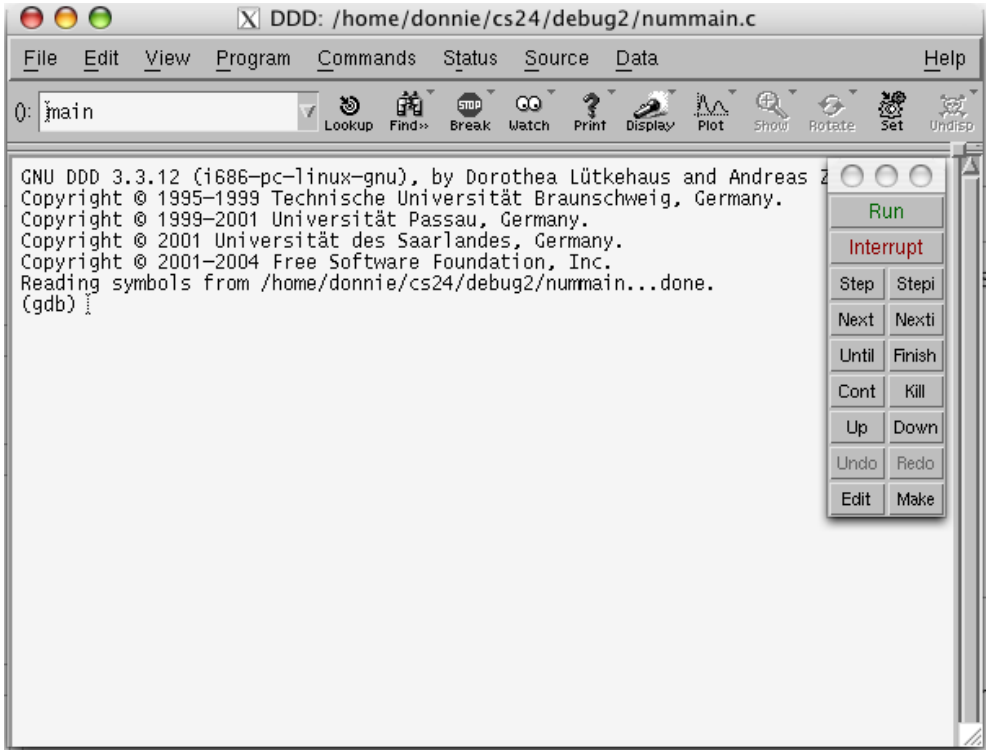
- All commands can be abbreviated with one character:

Cmd	Abbrev	Description
run	r	Starts program from main()
break	b	Set a breakpoint
continue	cont, c	Continues execution of program
step	s	Steps to next line (steps into functions)
next	n	Steps to next line (steps over functions)
print	p	Print out various values
quit	q	Exits gdb

- Many other commands in GDB as well!
  - gdb has an extensive help facility
  - help h — Displays general help information
  - help *command* — Displays help info about *command*

DDD: Graphical Frontend to GDB

- ddd is a graphical frontend to gdb
  - (if you like such things)
  - Start with: ddd *program*



- Provides some basic hand-holding with gdb
- (I don't think it helps much, but whatever.)

GDB and IA32 Nuances

- GDB can also step through IA32 code
  - Everything is basically the same, except for print expressions!
- To print individual registers:
  - print $eax [not %eax!]
    - Shows contents of %eax register
  - print *(int *) $esp
    - Shows value stored at (%esp)
    - Note: must tell gdb what type is at the pointer!
  - print *(int *) ($esp + 4)
    - Shows value stored at 4(%esp)
- To display all registers at once:
  - info registers [or: i reg]
    - Lists all registers (eax, ebx, …, eip, eflags), in base-10 and base-16

GDB Summary

- gdb is very good for watching your program's state and behavior as it executes
- Extensive set of commands
  - Will post a gdb cheat-sheet on the Moodle for you to use

- Isn't the best for finding dynamic memory mgmt bugs:
  - Accessing memory that you freed
  - Freeing a memory block multiple times
  - Accessing memory through an uninitialized pointer

Valgrind

- Valgrind is a suite of dynamic analysis tools built on a single unified platform
  - Simulates execution of Intel IA32 programs in a virtual machine
  - Allows sophisticated debugging of runtime behavior
  - Also allows sophisticated performance analysis of behavior
- Valgrind was designed to run on Linux
  - A port to MacOSX exists, but doesn't always work very well ☹
  - No valgrind port to Cygwin ☹
- Most widely used tool is the memcheck tool
  - Invoked by default when you run valgrind at command-line
- Valgrind also benefits from debugging symbols!
  - Always want to compile with –g –O0 when using valgrind

Valgrind (2)

- Usually use valgrind after you have identified and resolved other issues with gdb
  - gdb and valgrind are really good at different things
- Simple to use:

```
[user@host]> valgrind ./nummain
==30421== Memcheck, a memory error detector
==30421== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==30421== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright info
==30421== Command: ./nummain
==30421==
```

- Valgrind begins executing your program!
  - Prints out memory management issues as it runs

Memory Management Errors

- If your program does anything bad, valgrind immediately prints out an error message
- Example: nummain accesses previously-freed memory

```
Cleaning up list.
==30421== Invalid read of size 4
==30421==    at 0x8048595: empty (numlist.c:31)
==30421==    by 0x80487DC: main (nummain.c:41)
==30421==  Address 0x41bb224 is 4 bytes inside a block of size 8 free'd
==30421==    at 0x4026996: free (in /usr/lib/valgrind/…memcheck-x86-linux.so)
==30421==    by 0x8048591: empty (numlist.c:30)
==30421==    by 0x80487DC: main (nummain.c:41)
```

- As with gdb, valgrind tells you exactly what happened, and where it happened

Memory Management Errors (2)

- At end of program execution, valgrind prints out a summary of the program's heap usage
  - This output identifies leaked memory blocks!
- Example: nummain doesn't call free() on list nodes

```
Cleaning up list.
==30484==
==30484== HEAP SUMMARY:
==30484==     in use at exit: 80 bytes in 10 blocks
==30484==   total heap usage: 10 allocs, 0 frees, 80 bytes allocated
==30484==
==30484== LEAK SUMMARY:
==30484==    definitely lost: 8 bytes in 1 blocks
==30484==    indirectly lost: 72 bytes in 9 blocks
==30484==      possibly lost: 0 bytes in 0 blocks
==30484==    still reachable: 0 bytes in 0 blocks
==30484==         suppressed: 0 bytes in 0 blocks
==30484== Rerun with --leak-check=full to see details of leaked memory
```

Memory Management Errors (3)

- Rerunning with --leak-check=full causes valgrind to record where leaked blocks were allocated

```
[user@host]> valgrind --leak-check=full ./nummain
…
Cleaning up list.
==30485==
==30485== HEAP SUMMARY:
==30485==     in use at exit: 80 bytes in 10 blocks
==30485==   total heap usage: 10 allocs, 0 frees, 80 bytes allocated
==30485==
==30485== 80 (8 direct, 72 indirect) bytes in 1 blocks are definitely lost …
==30485==    at 0x402760A: malloc (in /usr/lib/valgrind/…-x86-linux.so)
==30485==    by 0x80484E7: add_num (numlist.c:12)
==30485==    by 0x8048722: main (nummain.c:29)
==30485==
==30485== LEAK SUMMARY:
…
```

GDB and Valgrind

- GDB and Valgrind are very powerful debugging tools!
- GDB is best for tracking down crashing bugs and other subtle defects
  - Primarily useful when need to watch what the program is doing as the bug occurs
  - Doesn't necessarily identify more subtle memory management issues
- Valgrind is best for finding subtle memory mgmt issues
  - Forgetting to free memory, freeing memory multiple times, accessing memory after freeing it, etc.
- Always use the right tool for the job!