Using the Debugger

Michael Jantz

Dr. Prasad Kulkarni

Debugger

- What is it
  - a powerful tool that supports examination of your program during execution.
- Idea behind debugging programs.
  - Creates additional symbol tables that permit tracking program behavior and relating it back to the source files
- Some common debuggers for UNIX/Linux
  - gdb, sdb ,dbx etc.

GDB

gdb is a tool for debugging C & C++ code

Some capabilities:

- run a program
- stop it on any line or the start of a function
- examine various types of information like values of variables, sequence of function calls
- stop execution when a variable changes
- change values of variables (during execution)
- call a function at any point in execution

Compilation for gdb

- Code must be compiled with the **-g** option
  - gcc -g -o file1 file1.c file2.c file3.o
- Which files can you debug?
  - You can debug file1.c, file 2.c, (not file3.o)
- Optimization is not always compatible
  - Due to optimizations which rearrange portions of the code

Building and Testing *bash*

1) Untar and navigate to the bash-4.2 directory:

    > tar xvzf eecs678-lab-gdb.tar.gz

    > cd gdb/bash-4.2

2) Configure *bash* for the build:

    > ./configure

3) Make *bash* using multiple jobs and with CFLAGS=-g

    > make -j8 CFLAGS=-g

4) Test the *bash* executable:

    > ./bash --version

Using GDB with *bash*

Starting GDB:

> gdb *program*

The build process created an executable file named *bash*

To start *bash* under GDB do:

gdb ./bash-4.2/bash

Breakpoints

- break (b)
  - Sets a breakpoint in program execution
  - tbreak (tb) temporary breakpoint. Exists until it is hit for the first time
- Breakpoint syntax
  - **b** *line-number*
  - **b** *function-name*
  - **b** *line-or-function* **if** *condition*
  - **b** *filename: line number*
- info breakpoints – Gives information on all active breakpoints
- delete (d) – Deletes the specified breakpoint number (e.g., d 1)

A Breakpoint in *bash*

- *bash* is a shell program. It provides a convenient command line interface to the OS, interprets commands, sets up pipelines, manages multiple jobs, etc.
- Debugging a running instance of *bash* can help you learn how the shell operates
- As an example, say you want to learn about how *bash* handles and executes commands. Place a breakpoint at the *execute_command* function:

```
gdb> b execute_command
```

Running *bash* Under GDB

run (r) - runs the loaded program under GDB

Can also specify arguments and I/O redirection now, e.g:

```
gdb> r arg1 arg2 < input > output
```

In our case, we can run a script with our *bash* executable:

```
gdb> r ./finder.sh bash-4.2/ execute 20
```

finder.sh

```
find $1 -name '*'.[ch] | xargs grep -c $2 | sort -t : +1.0 -2.0 --numeric --reverse | head --lines=$3
```

- find $1 -name '*'.[ch] – Find files with .c and .h extensions under the directory given by the first argument.
- xargs grep -c $2 – Search the set of files on standard input for the string given by the second argument. -c says that instead of printing out each usage in each file, give me the number of times $2 is used in each file.
- sort – Sort standard input and print the sorted order to standard output. -t : +1.0 -2.0 says sort using the second column on each line (delimited by the ':' character) as a key. --numeric says to sort numerically (as opposed to alphabetically). --reverse says sort in reverse order.
- head – print only the first *n* lines of standard input. --lines=$3 lets us set the number of lines with the third argument.

Common GDB Commands

- When you hit the breakpoint you should see:

  Breakpoint 1, execute_command (command=0x724088) at execute_cmd.c:376

- Now, gdb has stopped execution of *bash*. Try the following commands:
  - list (l) will list the source code around where execution has stopped. Alternatively, l *n,m* will display the source code in between two given line numbers *n* & *m*.
  - backtrace (bt) prints a backtrace of all stack frames. From this, you can tell how you got to where you are from the main. The output here says you are in execute_command, which was called from reader_loop, which was called from the main entry point.

Using the Frame Stack

- GDB currently has the execute_command frame selected. Use the info command to list information about the frame
  - info args – print the arguments passed into this frame
  - info locals – print the local arguments for this frame
  - help info – shows you everything info can tell you
- Additionally, print information about other stack frames using
  - up [n] – Select the frame n levels up in the call stack (towards main). n=1 if not specified.
  - down [n] – Select the frame n levels down in the call stack (you must have used up in order to come back down)
  - After you select a new frame, use info as described above to display information about the frame

Control Flow

- continue (c)
  - Continue until the next breakpoint is reached, the program terminates, or an error occurs (Don't use this just yet, we've got a few more commands to try).
- next (n)
  - Execute one instruction. Step over function calls.
- step (s)
  - Execute one instruction. Step into function calls.
- kill (k)
  - Kills the program being debugged (does not exit gdb – preserves everything else from the session, i.e., breakpoints.)

Inspecting and Assigning

- Continue to the end of the execute_command function:
  - finish (fin) – continue to the end of the function you're currently broken in
- Now, if you read the code in this function, it calls execute_command_internal with the current *command*. To look at *command*'s properties (or any object's) use the following:
  - print (p) *t* – Prints the value of some variable
  - whatis *t* – Prints the type of *t*
  - ptype *t* – Prints fields for the type *t*

Inspecting and Assigning (cont.)

- Try these commands to inspect the command object:
  - gdb> whatis *command* – tells us the type of *command*.
  - gdb> ptype *command* – displays all the fields the command type
  - gdb> p command->value – prints the value of *command->value*

- You can also assign values in gdb:
  - gdb> set var command=0x0 – sets the *command* pointer to 0x0

Printing Examples

(gdb) p command
$14 = (COMMAND *) 0x724088

(gdb) ptype command

```
type = struct command {
    enum command_type type;
    int flags;
    int line;
    REDIRECT *redirects;
    union {
        struct for_com *For;
        ...
        struct coproc_com *Coproc;
    } value;
} *
```

Printing Examples

```
(gdb) p command->type
$15= cm_connection

(gdb) p (struct connection *) command->value
$16 = (struct connection *) 0x724048

(gdb) ptype ((struct connection *) command->value)
type = struct connection {
    int ignore;
    COMMAND *first;
    COMMAND *second;
    int connector;
} *
```

Printing Examples

(gdb) p ((struct connection *) command->value)->first
$17 = (COMMAND *) 0x721108

(gdb) p ((struct connection *) command->value)->first->type
$18 = cm_simple

(gdb) ptype ((struct simple_com *) ((struct connection *) command->value)->first)

```
type = struct simple_com {
    int flags;
    int line;
    WORD_LIST *words;
    REDIRECT *redirects;
} *
```

Printing Examples

(gdb) p ((struct simple_com *) ((struct connection *) command->value)->first)->words
$19 = (WORD_LIST *) 0xdfdfdfdfdfdfdfdf

(gdb) ptype ((struct simple_com *) ((struct connection *) command->value)->first)->words

```
type = struct word_list {
    struct word_list *next;
    WORD_DESC *word;
} *
```

(gdb) ptype ((struct simple_com *) ((struct connection *) command->value)->first)->words->word

```
type = struct word_desc {
    char *word;
    int flags;
} *
```

(gdb) p ((struct simple_com *) ((struct connection *) command->value)->first)->words->word
Cannot access memory at address 0xdfdfdfdfdfdfdfe7

Calling Functions from GDB

- The *call* command allows you to call other functions in your code within GDB.
- Very useful for printing complicated data structures within the debugger

Call Example

```
(gdb) b execute_simple_command
Breakpoint 2 at 0x4380a4: file execute_cmd.c, line 3650.
(gdb) c
Continuing.

(gdb) p simple_command
$28 = (SIMPLE_COM *) 0x721148

(gdb) p simple_command->words
$29 = (WORD_LIST *) 0x721fe8

(gdb) call _print_word_list(simple_command->words, " ", printf)
(gdb) call fflush(stdout)
find $1 -name '*'.[ch]$30 = 0
```

GDB References

- The Unix manual is a good quick reference for common GDB commands:
  - At a terminal, type: man gdb
- While running GDB, help will give you any information you need for any command:
  - help (h) *command*
- If you need to do some heavy lifting with GDB, the official documentation for users is at this website:
  - http://sourceware.org/gdb/current/onlinedocs/gdb_toc.html

Backup

Multiple Threads

- GDB also has a set of commands for finer control of multi-threaded applications.
- GDB comes with these commands for controlling multiple threads:
  - info threads – Print a numbered list of all current threads and their contexts. An asterisk denotes the thread on which GDB is currently focused.
  - thread <thread #> - Switch focus to the thread numbered <thread #>.
  - thread apply (all | <thread # list>) cmd – Apply cmd to all threads or each thread in the <thread # list>.
- For example, *thread apply all bt* shows the stack trace for each thread.

Automatic Source Navigation

- GDB comes with a tool for automatic source navigation called the Text User Interface (TUI).
- To access the TUI, do C-x, C-a in the shell running GDB.
- It should split the terminal. Now, when you run your program under GDB, the source will be displayed in the screen above your command line.
- To switch between control of the the source code screen and the command line do: C-x, o.
- Alternatively, if you would like a more graphical user interface, you can use the DDD debugger (which is essentially identical to the TUI, but provides more buttons and mouse over actions).

Valgrind

- A heavyweight tool for dynamically catching hard-to-detect errors in programs.
  - Only checks code that it runs
- Valgrind is a virtual machine and quite a bit slower than normally execution.
  - Translates binaries into an intermediate representation performs some modifications and recompiles the transformations at runtime as needed.
  - Allows it to instrument code with safe guards profiling tools.
- Valgrind's main tool is a memory checker that detects memory errors in programs
- Also has a few other useful tools such as cache and branch miss-prediction profilers, a heap profiler, a multi-thread data race detector, and more.

Valgrind

- Memory checker (memcheck) helps find:
  - Memory leaks
  - Usage of uninitialized variables
    - Often the cause of non-deterministic behavior in single threaded software.
  - Bad frees of heap blocks
    - Double frees
    - Mismatched frees (frees on addresses without associated mallocs)
  - Accesses to unallocated memory
    - Accesses to invalid stack and heap addresses
      - Freed heap addresses
      - Out of bounds heap addresses
      - Out of bounds stack addresses

Valgrind - Memcheck

- To run valgrind use:
  - “valgrind [optional flags] <executable>”
- By default only the memcheck tool is used.

```c
#include <stdlib.h>

void* still_reachable;
void* possibly_lost;

int main() {
  int uninitialized_variable; // This variable is never given a value.

  for (; uninitialized_variable < 100; uninitialized_variable++) {
    void** definitely_lost = (void**) malloc(sizeof(void*)); // allocate a
                                                             // pointer on the
                                                             // heap.

    *definitely_lost = (void*) malloc(7); // Give the pointer something else to
                                          // point to on the heap. This will be
                                          // indirectly lost.
  }

  // At this point, definitely_lost is out of scope and we can no longer free
  // it. The pointer pointed to by definitely_lost is indirectly lost since we
  // were only able to reach the pointer through definitely_lost.

  still_reachable = malloc(42); // This value is never freed but is pointed to
                                // in the global scope at program completion.

  possibly_lost = malloc(10);
  possibly_lost += 4; // This is similar to still reachable except there is a
                      // pointer pointing to the middle of the allocated block
                      // but nothing points to the front of the block. This is
                      // very odd behavior and usually is a memory leak (but not
                      // always).
}
```

```
[jrobinson@localhost Development]$ valgrind ./test
==2606== Memcheck, a memory error detector
==2606== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==2606== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==2606== Command: ./test
==2606==
==2606== Conditional jump or move depends on uninitialised value(s)
==2606==    at 0x40056A: main (test.c:9)
==2606==
==2606==
==2606== HEAP SUMMARY:
==2606==     in use at exit: 1,553 bytes in 202 blocks
==2606==   total heap usage: 202 allocs, 0 frees, 1,553 bytes allocated
==2606==
==2606== LEAK SUMMARY:
==2606==    definitely lost: 800 bytes in 100 blocks
==2606==    indirectly lost: 700 bytes in 100 blocks
==2606==      possibly lost: 10 bytes in 1 blocks
==2606==    still reachable: 43 bytes in 1 blocks
==2606==         suppressed: 0 bytes in 0 blocks
==2606== Rerun with --leak-check=full to see details of leaked memory
==2606==
==2606== For counts of detected and suppressed errors, rerun with: -v
==2606== Use --track-origins=yes to see where uninitialised values come from
==2606== ERROR SUMMARY: 101 errors from 1 contexts (suppressed: 0 from 0)
```

valgrind_test.c – a bad program that compiles without warnings (gcc -Wall -g valgrind_test.c)

Valgrind shows all of the problems with this code

Valgrind - Memcheck

- Rerun with “--leak-check=full” and “--track-origins=yes” as suggested by memcheck.

```
[jrobinson@localhost Development]$ valgrind --leak-check=full --track-origins=yes ./test
==16850== Memcheck, a memory error detector
==16850== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==16850== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==16850== Command: ./test
==16850==
==16850== Conditional jump or move depends on uninitialised value(s)
==16850==    at 0x40056A: main (test.c:9)
==16850==  Uninitialised value was created by a stack allocation
==16850==    at 0x400536: main (test.c:6)
==16850==
==16850==
==16850== HEAP SUMMARY:
==16850==     in use at exit: 1,552 bytes in 202 blocks
==16850==   total heap usage: 202 allocs, 0 frees, 1,552 bytes allocated
==16850==
==16850== 10 bytes in 1 blocks are possibly lost in loss record 1 of 4
==16850==    at 0x4C28C50: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==16850==    by 0x400586: main (test.c:26)
==16850==
==16850== 1,500 (800 direct, 700 indirect) bytes in 100 blocks are definitely lost in loss record 4 of 4
==16850==    at 0x4C28C50: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==16850==    by 0x400549: main (test.c:10)
==16850==
==16850== LEAK SUMMARY:
==16850==    definitely lost: 800 bytes in 100 blocks
==16850==    indirectly lost: 700 bytes in 100 blocks
==16850==      possibly lost: 10 bytes in 1 blocks
==16850==    still reachable: 42 bytes in 1 blocks
==16850==         suppressed: 0 bytes in 0 blocks
==16850== Reachable blocks (those to which a pointer was found) are not shown.
==16850== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==16850==
==16850== For counts of detected and suppressed errors, rerun with: -v
==16850== ERROR SUMMARY: 103 errors from 3 contexts (suppressed: 0 from 0)
```

Valgrind - Memcheck

- Rerun with “--show-leak-kinds=all” appended to the options to see the still reachable blocks.
- Valgrind is not perfect.
  - It will only catch errors that only occur during a program run.
    - If an error inducing code is not reached or specific input conditions that trigger an error are not met, then Valgrind will not detect it in that run.
    - Importance of thorough unit tests.
  - Custom memory allocators can confuse Memcheck.