CS 331

Spring 2016

GDB and x86-64 Assembly Language

I am not a big fan of debuggers for high-level languages. While they are undeniably useful at times, they are no substitute for careful coding. If code is well structured, I usually find it easier to debug analytically (i.e, by staring at it) than by running it through a debugger. In any event, monkey-coding with a debugger is still monkey-coding, and I don't want to fly in an airplane being guided by monkey-code.

The situation in assembly language is quite different. Assembly instructions can be opaque, especially if you aren't used to programming in assembly language. In addition, I/O at the assembly level is quite complex; printing values at various places in your code is often not an option.. Debuggers are frequently the only way to discover what your code is actually doing. There are nicer assemblers than the gas (gnu assembler) that we will use, but the gas-gdb combination works quite well and is easy to use.

In this document we'll go briefly through the essential commands for gdb. There is more complete documentation available on any of the Linux systems; just type

```
info gdb
```

And, of course, there is no end of documentation online.

To assemble your program with hooks for gdb use the hyphen g flag. For example, if your assembly language program is foobar.s you would assemble it wth

```
gcc foobar.s –g –o foobar
```

and then run gdp with

```
gdb foobar
```

You should always have the program visible in an editor that includes line numbers while you run gdb.

Alternatively, you can run gdb in a "graphical mode" with

```
gdb –tui foobar
```

If you use the graphical mode you get two windows, one with your assembly code and the other a "command" window into which you can type. If you click in the code window you can use the arrow keys to scroll around in the code. When you enter commands that execute lines of code (such as "step") the code window shows the region of code around which you are currently executing.

Your first step in gdb will be to set one or more breakpoints. The command is

```
break <line number>
```

as in

```
break 35
```

The run command starts the program; it will execute up to, but not including, the line with the first breakpoint. The text of the next line to be executed will be printed; I don't find this much help unless I have a copy of the program at hand during the session.

When the program is stopped you can examine the memory. The print commands are useful for registers:

```
print/d $rax
```

prints the contents of register rax as a decimal number;

```
print/x $rsp
```

prints the contents of rsp in hexadecimal.

In graphical mode you can open a window that will show the contents of all of the registers:

```
layout re
```

shows the registers; if this opens with the wrong registers visible

```
tui reg general
```

will focus the register window on the general purpose registers that we use for compilation. Each register is printed both in hex and decimal format.

Values on the stack are most easily viewed with the

```
x/ NFU  <address>
```

command. Here N is the number of consecutive locations to be printed, F is the format, which can be ‘d’ for decimal ‘x’ for hex, ‘t’ for binary and ‘c’ for ASCII, and U is the size of each location, which can be ‘b’ for byte, ‘h’ for a 16-bit half-word, or ‘w’ for a 32-bit word. For example

```
x/10dw  8+$rsp
```

will print 10 values as integers starting 8 bytes below the stack pointer.

If you are halted at a breakpoint,

```
cont
```

will continue the execution up to the next breakpoint, or the end of the program. Alternatively, you can step through the program a block of instructions at a time:

```
stepi 5
```

executes the next 5 instructions (if you omit the number, it defaults to 1);

```
nexti 5
```

does the same thing, though it steps over rather than into function calls.

Finally,

```
quit
```

terminates the debugging session.

Here is a short table of gdb commands

Command	Shorthand	Examples	Description
help [command]		help break	Gives information about the command
quit	q	quit	Exit from gdb
run	r	run	start the program; execution continutes up to a breakpoint
cont	c	cont	Continue execution from the current point, until the next breakpoint is reached
stepi <n>	si <n>	stepi 5 stepi	Execute the next 5 instructions, stepping into function calls. Note that <n> defaults to 1
nexti <n>	ni <n>	nexti 5 nexti	Execute the next 5 instructions, stepping over function calls. Note that <n> defaults to 1.
break <address>	b <add>	break 34	Sets a breakpoint. The <address> may either be a label in your program or a line number.
info break		info break	Gives a numbered list of all current breakpoints.
delete <n>	d <n>	delete 3	Deletes the nth breakpoint. If you omit the n all breakpoints will be deleted.
print <expression>	p/d <exp>	print/d $rax print/x $rsp print/t $rsp	Print the value of the expression. You can modify the command with formats: /d for decimal integers, /x for hexadecimal and /t for binary.
x/NFU <address>		x/4dw -16+$rsp	Print N consecutive values in format F of words of size U. N should be an integer, F either ‘d’ for decimal, ‘x’ for hex, ‘t’ for binary, and ‘c’ for ASCII, and U should be ‘b’ for byte, ‘h’ for 16-bit half-word, and ‘w’ for 32-bit word.
display <expression>		display $rax	Registers the expression as something to be printed each time the execution is halted.
info display		info display	Gives a numbered list of all of the current display expressions
undisplay <n>		undisplay 3	Removes one of the current display expressions