Faculty of Science

Course:	CSCI 2050U: Computer Architecture I
Topic:	Debugging with `gdb`

Overview

The purpose of this document is to give you skills in `gdb`, which is the most popular debugger on the Linux platform (as well as several other platforms). `gdb` is feature-packed, but it its interface is entirely text-based.

Part 1 – Getting Started

Let's get our assembly language program ready for debugging. You need to assemble project with the `-g` (and/or the `-ggdb`) flag. We'll start with the following assembly language program (`debug.asm`) for this guide:

```
extern printf
global main

section .text
main:
      mov rdi, format       ; argument #1
      mov rsi, message      ; argument #2
      mov rax, 0
      call printf           ; call printf

      mov rax, 0
      ret                   ; return 0

section .data
      message:    db "Hello, world!", 0
      format:     db "%s", 0xa, 0
      courseCode: dq 2050
```

Below, we assemble and link the program (with the appropriate flags) so that we can use `gdb` on the resulting executable:

```
$ nasm -f elf64 -g -F dwarf -o debug.o debug.asm
$ gcc -m64 -o debug debug.o
$ gdb debug
```

We are now debugging our application.

Part 2 – Execution Commands

Like most debuggers, `gdb` will let us run our program, stopping at breakpoints, and even step through our program line-by-line. To set a breakpoint at the start of the `main` function, use the `break` (or `b` for short) command:

```
(gdb) b main
Breakpoint 1 at 0x400530: file debug.asm, line 6.
```

You can set a breakpoint at any label in an assembly language program. You can also set a breakpoint at any line in the original source file:

```
(gdb) b debug.asm:9
Breakpoint 2 at 0x400549: file debug.asm, line 9.
```

We can now run our program with the `run` (or `r` for short) command:

```
(gdb) r
Starting program: /home/rfortier/Documents/Winter 2017/CSCI
2050u/AssemblyExamples/using_gdb/debug

Breakpoint 1, main () at debug.asm:6
6           mov   rdi, format     ; argument #1
```

Notice that `gdb` stopped at our breakpoint. We can continue running until the next breakpoint (line 9) using the `continue` (or `c` for short) command:

```
(gdb) c
Continuing.

Breakpoint 2, main () at debug.asm:9
9           call printf       ; call printf
```

We can also step through our program line-by-line, using the command `next` (or `n` for short):

```
(gdb) next
Hello, world!
11          mov rax, 0
```

`gdb` shows us the next line of code, but we can see more context using the `list` (or `l` for short) command:

```
(gdb) list
6           mov rdi, format       ; argument #1
7           mov rsi, message      ; argument #2
8           mov rax, 0
9           call printf           ; call printf
10
11          mov rax, 0
12          ret                   ; return 0
13
14    section .data
15          message:    db "Hello, world!", 0
```

Part 3 – Data Commands

When debugging our program, we are probably going to want to know the state of our registers and variables. Without being able to do so, it will be challenging for us to identify where logic errors happen in our program. The easiest way to view the contents of a variable is using the `print` (or `p` for short) command:

```
(gdb) p courseCode
$1 = 2050
```

It is also possible to print using type specifiers, and control how your output is displayed. A comprehensive set of type specifiers is given in the table, below:

Specifier	Meaning
`t`	binary (base [t]wo)
`o`	[o]ctal
`x`	he[x]adecimal
`a`	[a]ddress (hexadecimal absolute, plus hexadecimal offset from a close label)
`c`	[c]haracter
`s`	[s]tring
`d`	signed [d]ecimal
`u`	[u]nsigned decimal
`f`	[f]loating point

Examples of usage:

```
(gdb) p/x courseCode
$1 = 0x802
(gdb) p/t courseCode
$4 = 100000000010
(gdb) print/c message
$3 = 72 'H'
```

There is also the x (e[x]amine) command for viewing memory contents. This is useful for strings and arrays:

```
(gdb) x &message
0x601040 <message>:     "Hello, world!"
```

The & in the above command has the same meaning as in C/C++: "the address of". This command has options similar to the `print` command. In general, the format of the command is:

`x/nfu address`

- n – how many of each data unit
- f – what type specifier (same as with print, but `i` is also possible for instructions)
- u – unit (data unit size)

Data unit sizes are given in the table below:

Data Unit Size	Meaning
`b`	[b]ytes
`h`	[h]alf words (words in x64 parlance)
`w`	[w]ords (double words or dwords in x64 parlance)

`g`	[g]iant words (quad words or qwords in x64 parlance)

Sample usage:

```
(gdb) x/5cb &message
0x601040 <message>:     72 'H'      101 'e'     108 'l'     108 'l'     111 'o'
(gdb) x/2s &message
0x601040 <message>:     "Hello, world!"
0x60104e <format>:      "%s\n"
(gdb) x/2dg &courseCode
0x601052 <courseCode>:  2050  0
(gdb) x/2tg &courseCode
0x601052 <courseCode>:
      0000000000000000000000000000000000000000000000000000100000000010
      0000000000000000000000000000000000000000000000000000000000000000
(gdb) x/2xg &courseCode
0x601052 <courseCode>:  0x0000000000000802      0x0000000000000000
```

Another thing we are likely to want to do is to view the registers. You can view the normal (integer) registers with the `info registers` command, and the floating point registers with the `info float` command:

```
(gdb) info r
rax            0xe      14
rbx            0x0      0
rcx            0xd      13
rdx            0x7ffff7dd59e0 140737351866848
rsi            0x7ffffff2     2147483634
rdi            0x1      1
rbp            0x0      0x0
rsp            0x7fffffffda88 0x7fffffffda88
r8             0xffffffff     4294967295
r9             0x0      0
r10            0x7ffff7dd26a0 140737351853728
r11            0x246    582
r12            0x400440 4195392
r13            0x7fffffffdb60 140737488345952
r14            0x0      0r15            0x0     0
rip            0x40054e 0x40054e <main+30>
eflags         0x212    [ AF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb) info float
  R7: Empty   0x00000000000000000000
  R6: Empty   0x00000000000000000000
  R5: Empty   0x00000000000000000000
  R4: Empty   0x00000000000000000000
  R3: Empty   0x00000000000000000000
  R2: Empty   0x00000000000000000000
  R1: Empty   0x00000000000000000000
```

```
=>R0: Empty   0x00000000000000000000

Status Word:         0x0000
                       TOP: 0
Control Word:        0x037f   IM DM ZM OM UM PM
                       PC: Extended Precision (64-bits)
                       RC: Round to nearest
Tag Word:            0xffff
Instruction Pointer: 0x00:0x00000000
Operand Pointer:     0x00:0x00000000
Opcode:              0x0000
```

Finally, it might be useful to know how to quit gdb:

```
(gdb) quit
A debugging session is active.

      Inferior 1 [process 7900] will be killed.

Quit anyway? (y or n) y
```

References

[1] https://linux.die.net/man/1/gdb