

In lecture we have briefly discussed assembly language, and how you might reverse engineer (disassemble) an executable to see the assembly-level instructions.

The Objectives of this lab are the following:

- Inspect some of the architecture elements of the CS linux computers
- Become familiar with GDB, which is the standard (plain text) debugger available on most linux-ish computers
- Use the disassemble feature of gdb to inspect the assembly code equivalent of a program

As was the case for labs 1 and 2, ask the TAs lots of questions. They are there to help. And, **you CAN work with peers in completing the labs. This is a bit tricky when everybody is remote; you may use slack, email, discord, etc. Up to you.** But, in no circumstances should you SHARE code, which involves emailing code and then submitting it as your own.

Submitting your work

Submit the asked-for file as the Lab 3 Submission item on Canvas. You must submit your file by the due date/time specified on Canvas.

Logging into your CS account

Use `ssh` to connect to one of the CS linux computers. All instructions in this lab – referring to `gdb`, `gcc`, etc. – assume the use of the CS machines.

I. Architecture specifics

In this part of the lab, you'll explore some of the architecture elements of the computer that you are logged into. First, let us inspect if the CS machines are 32-, or 64-bit machines. There are many ways to find this out; only a few are shown here.

The `uname` linux command prints system information. Issue `uname`, with the `-a`(ll) flag:

```
$uname -a
```

You'll see a list of details, including the name of the machine, the processor type, the operating system, etc.

Next, let us inspect the hardware and memory attributes of the computer that you've logged onto. One such command to get that information is the `free` command, which can be issued with a variety of flags, including `m`(egabytes), `h`(uman readable), `g`(igabytes), etc. Issue each of the following (they provide the same info, but different formatting) to determine how much memory is available on the linux machine where you are logged on.

```
$free -g
```

```
$free -m
```

```
$free -h
```

The program `who` and `htop` inform you who else is logged onto the computer, and which processes are currently running – and what resources they are using.

Issue `who`, to see who is logged on. Your username should be among those listed.

```
$who
```

Although not the focus of this class (instead, you'll learn more about this when you take CSCI 447, Operating Systems), there are a variety of command line programs that you can use to inspect what is running – and what memory resources each program is using – on your computer.

Issue `htop`, to see who is logged on, what the load is on each of the CPUs, how much memory each process is using, etc. The `htop` program includes help (F1), and a variety of other options.

```
linux.cs.wvu.edu - PuTTY
```

```

 1 [          0.0%]  5 [          0.0%]
 2 [          0.0%]  6 [          0.0%]
 3 [          0.0%]  7 [          0.0%]
 4 [|||        2.6%]  8 [|         1.3%]
Mem[|||||1.99G/15.6G] Tasks: 779, 840 thr: 1 running
Swp[|||||2.04G/14.9G] Load average: 0.20 0.11 0.08
                          Uptime: 97 days, 23:39:32

  PID USER      PRI  NI  VIRT   RES   SHR  S  CPU% MEM%   TIME+  Command
32492 jagodzif  20   0 27604  4784  3220  R   1.9   0.0   0:02.40 htop
 2141 gdm        20   0  887M 30300 1508  S   0.6   0.2 8h01:01 /usr/lib/gnome-settings-daemon/gsd-col
31659 taylo230  20   0 1298M 245M 31456  S   0.6   1.5 0:04.95 /home/taylo230/.vscode-server/bin/cd9e
1550 kernoops   20   0 56940   288   180  S   0.6   0.0 6:52.28 /usr/sbin/kerneloops --test
12559 taylo230  20   0 1406M  9584  8516  S   0.0   0.1 4:01.61 /home/taylo230/.vscode-server/extension
32553 covinga    20   0  348M 58728 14680  S   0.0   0.4 0:02.01 /usr/bin/emacs --daemon
32371 taylo230  20   0 1407M 11008  8532  S   0.0   0.1 5:00.98 /home/taylo230/.vscode-server/extension
 2306 taylo230  20   0  682M 11136  7008  S   0.0   0.1 0:23.32 /home/taylo230/.vscode-server/extension
12549 taylo230  20   0 1406M  9584  8516  S   0.0   0.1 4:55.25 /home/taylo230/.vscode-server/extension
 1041 root       20   0 1099M  7884  1640  S   0.0   0.0 14:33.33 /usr/sbin/nsd
10273 cacerem    20   0  296M    0     0  S   0.0   0.0 2:19.26 /usr/lib/gnome-online-accounts/goa-ide
 2357 taylo230  20   0 1558M 10100  8736  S   0.0   0.1 2:02.75 /home/taylo230/.vscode-server/extension
24981 taylo230  20   0 1423M 12440  8808  S   0.0   0.1 1:43.57 /home/taylo230/.vscode-server/extension
13615 taylo230  20   0 1407M 10388  8476  S   0.0   0.1 4:34.93 /home/taylo230/.vscode-server/extension
28246 sweenef    20   0  296M    0     0  S   0.0   0.0 2:21.01 /usr/lib/gnome-online-accounts/goa-ide
F1Help F2Setup F3Search F4Filter F5Free F6SortBy F7Nice F8Nice F9Kill F10Quit

```

II. The C-program

Now that we have a rudimentary understanding what type of computer you are logged into – which is important, if ever you needed to look at the instruction set to get a better understanding of the assembly opcodes that are available – let us start by writing and compiling a simple C program.

Save the C program into the file *sample.c*, shown right.

This seems harmless enough ... and it is. As mentioned in lecture, there are multiple ways that you can inspect the assembly code that would be generated from the source code file.

```
int main() {  
    int x, y;  
    x = 10;  
    y = 15;  
    x = y-x;  
    y = x+y;  
}
```

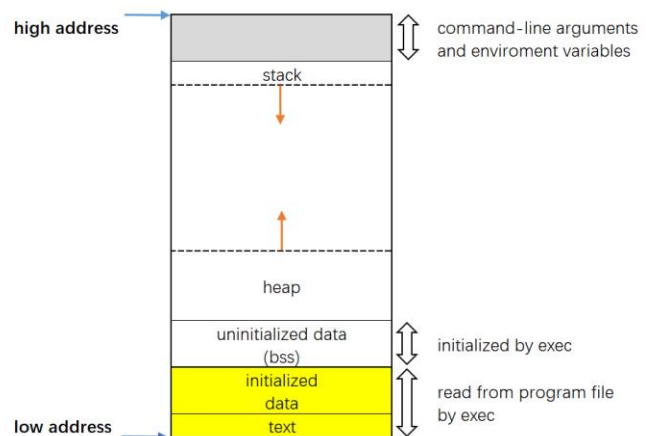
III. Using gcc to generate the assembly code

As was shown in the lecture slides, issue the following, to generate the .s file for the program *sample.c*:

```
$gcc -S sample.c
```

Let's inspect the assembly code, which is in the file *sample.s*.

```
pushq    %rbp  
movq     %rsp, %rbp  
movl     $10, -8(%rbp)  
movl     $15, -4(%rbp)  
movl     -4(%rbp), %eax  
subl     -8(%rbp), %eax  
movl     %eax, -8(%rbp)  
movl     -8(%rbp), %eax  
addl     %eax, -4(%rbp)  
movl     $0, %eax  
popq     %rbp  
ret
```



All lines that begin with a period are not being shown here, because they are annotations for the linker. Become familiar with the syntax and registers. **Do a web search and/or ask your TA for the questions to the following:**

- What does a % refer to?
- What does a \$ designate
- What do rbp, eax, and rsp refer to?
- What does a "-8" refer to, as for example -8(%rbp)
- What do the opcodes pushq, mov, sub, add, pop, ret specify
- What do the single letter suffixes l, q (and are there others?) specify, as for example movq versus movl.

IV. First use of gdb.

GDB is the GNU Debugger, which is a text-only portable debugger that is included with most Linux-ish operating systems. It provides all of the features that you'd find in any IDE debugger, such as setting breakpoints, stepping through functions, stepping into functions, etc.

Available in the files section of Canvas is a GDB debugger document. Although the gdb commands needed for this lab are all presented here, you will need to refer to the documentation for (probably) homework 2, and definitely project 2.

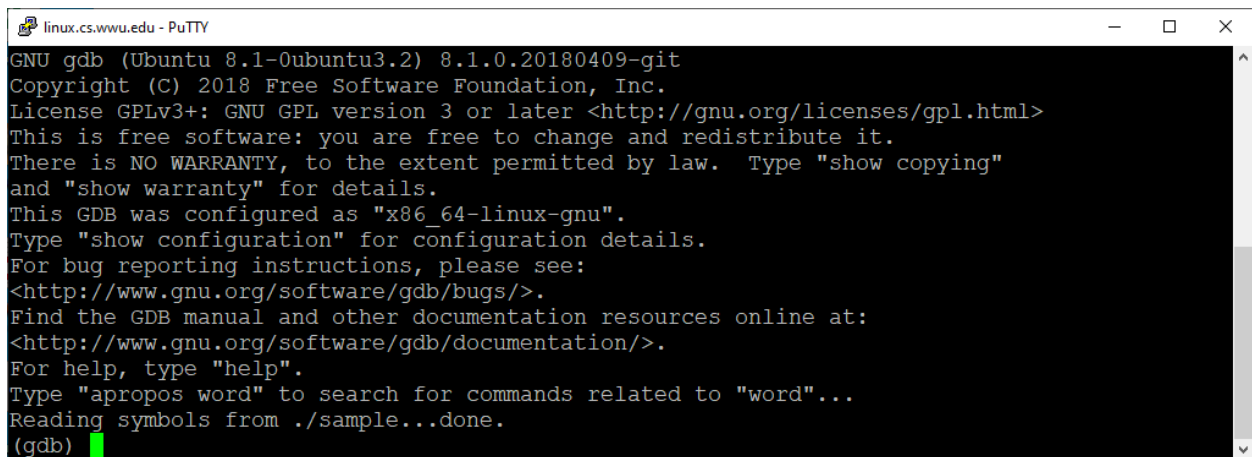
To use a program with gdb, you must compile the program with the `-g` flag, to inform the compiler to insert debugger symbols so that the executable can be intercepted by gdb to permit inspecting various elements of the program.

```
$gcc -g -o sample sample.c
```

Now that you have created an executable suitable for use in gdb, start gdb, and “load” the sample executable into gdb:

```
$gdb ./sample
```

You'll see the (not too stylish – hey I did say text-only!) gdb screen, similar to what is shown below. That is “the” gdb prompt.

A screenshot of a terminal window titled "linux.cs.wvu.edu - PuTTY". The terminal displays the GNU gdb startup screen for Ubuntu 8.1-0ubuntu3.2. It shows the version "8.1.0.20180409-git", copyright information for the Free Software Foundation, Inc., and the GNU GPL license. It also provides links to the GDB manual and documentation, and instructions on how to get help or report bugs. The prompt "(gdb)" is visible at the bottom of the screen, followed by a green cursor.

```
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./sample...done.
(gdb)
```

At the (gdb) prompt, set a breakpoint at the start of the `main()` function of the program by using the `b` command. This will make the debugger pause when it starts executing the `main()` function.

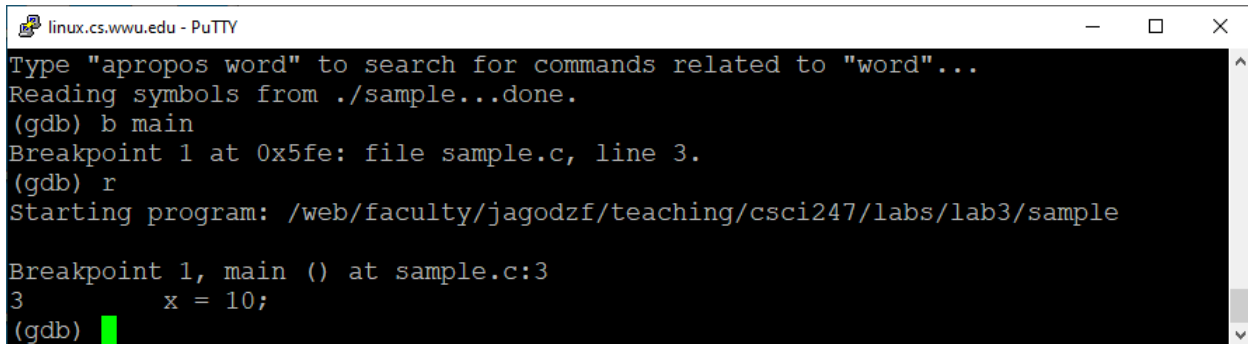
```
(gdb) b main
```

When issued, gdb will inform you that a breakpoint has been set.

At the (gdb) prompt, run the program by using the `r` command:

```
(gdb) r
```

You will see the output similar to what is shown in the following figure:



```
linux.cs.wvu.edu - PuTTY
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./sample...done.
(gdb) b main
Breakpoint 1 at 0x5fe: file sample.c, line 3.
(gdb) r
Starting program: /web/faculty/jagodzfi/teaching/csci247/labs/lab3/sample

Breakpoint 1, main () at sample.c:3
3      x = 10;
(gdb)
```

At this point, gdb has reached a breakpoint, and is now waiting for your command. There are a variety of commands (refer to the gdb tutorial). One of these commands is `next`, which you can use to proceed to the next line of code.

Two useful commands are `continue`, and `display`, where `display` allows you to inspect the value of any variable that is in scope. For example, assume you want to inspect the value of the variable `x`:

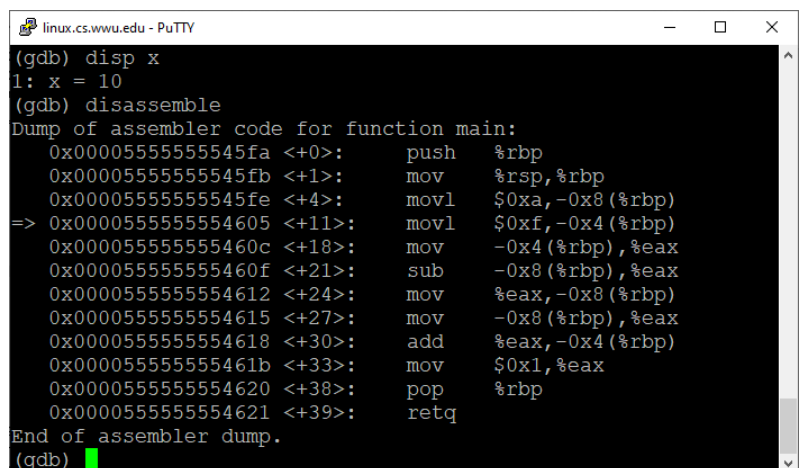
```
(gdb) disp x
```

V. Disassembling

GDB includes a `disassemble` option to convert the machine language of the program back to assembly language. Instruct gdb to disassemble the program (machine language) into assembly language, by issuing the `disassemble` command:

```
(gdb) disassemble
```

The output should be similar to what is shown in the figure on the right.



```
linux.cs.wvu.edu - PuTTY
(gdb) disp x
1: x = 10
(gdb) disassemble
Dump of assembler code for function main:
0x00005555555545fa <+0>:    push    %rbp
0x00005555555545fb <+1>:    mov     %rsp,%rbp
0x00005555555545fe <+4>:    movl    $0xa,-0x8(%rbp)
=> 0x0000555555554605 <+11>:   movl    $0xf,-0x4(%rbp)
0x000055555555460c <+18>:   mov     -0x4(%rbp),%eax
0x000055555555460f <+21>:   sub     -0x8(%rbp),%eax
0x0000555555554612 <+24>:   mov     %eax,-0x8(%rbp)
0x0000555555554615 <+27>:   mov     -0x8(%rbp),%eax
0x0000555555554618 <+30>:   add     %eax,-0x4(%rbp)
0x000055555555461b <+33>:   mov     $0x1,%eax
0x0000555555554620 <+38>:   pop     %rbp
0x0000555555554621 <+39>:   retq
End of assembler dump.
(gdb)
```

What similarities and differences are there between this disassembled code, and the assembly code that you generated using the `gcc -S` flag? For starters, notice that gdb is telling you, via the `=>`, where the program is currently executing. Neat!

Note that `%rbp` (the base pointer) is set to the top of the stack. The local variables `x` and `y` are stored on the stack at positions beyond the base pointer. Since `x` and `y` are both `int`, each requires 4 bytes (that's the C default) so their values are stored at addresses `-0x8(%rbp)` and `-0x4(%rbp)`, respectively, where the 8 and 4 refer to addresses BEYOND the start of the memory location referred to by `rbp`.

Here's the assembly instructions again, with annotations:

<code>push %rbp</code>	<code># save the old value of %rbp</code>
<code>mov %rsp,%rbp</code>	<code># set %rbp to the value of %rsp</code>
<code>movl \$0xa,-0x8(%rbp)</code>	<code># x = 10</code>
<code>movl \$0xf,-0x4(%rbp)</code>	<code># y = 15</code>
<code>mov -0x4(%rbp),%eax</code>	<code># load y into %eax</code>
<code>sub -0x8(%rbp),%eax</code>	<code># y - x is now in %eax</code>
<code>mov %eax,-0x8(%rbp)</code>	<code># x = y - x</code>
<code>mov -0x8(%rbp),%eax</code>	<code># load x into %eax</code>
<code>add %eax,-0x4(%rbp)</code>	<code># y = x + y</code>
<code>mov \$0x0,%eax</code>	
<code>pop %rbp</code>	<code># restore the old value of %rbp</code>
<code>retq</code>	<code># return from main()</code>

VI. Disassembling aProgram

For this part of the lab, you will retrieve an executable from my (Filip's) home directory, and disassemble it using `gdb`.

Retrieve the *aProgram* executable file from the following directory, on the linux CS machines:

```
/home/jagodzif/public_html/teaching/csci247/labs/lab3
```

Refer to lab 1 on how to copy a file from one directory to another.

Your task: Use `gdb` to set a breakpoint, and disassemble the program. Then, compose the file *aProgram.c*, which you will upload to Canvas.

Rubric and submission

Please upload your *aProgram.c* file to Canvas.

The <i>aProgram.c</i> file contains the plain text <code>.c</code> source code that would generate the assembly code for the executable <i>aProgram</i> .	10 points
	10 points