GDB 2

In this lab, we will be working on creating assembly files and Makefiles. We will also be working with `gdb` for a deeper understanding. After this discussion, you should be familiar with the following:

1. Creating a new assembly file.
2. Using instructions (`popcnt` and `add`).
3. Compiling assembly files using `as` and `ld`.
4. Creating a `Makefile` and adding recipes.
5. Using `gdb` commands (`set`, `break`, `stepi`) and working with registers (i.e., `$rax`, `$rxb`, `$rip`).

Setup

Follow the link posted on Piazza under Resources for the discussion GitHub Classroom link to create your lab repository.

Log into your UNIX environment, clone the repository and change directories into the repository working copy.

In this lab, we will use emacs to create several files. Don't forget you have to force emacs to save what you type into the file. To do this, you can use the menu, or more easily press the short-cut for saving Control-x followed by Control-s (`c-x c-s`).

Exercise 1: Creating Assembly Files

Intro

Assembly instructions provide a detailed description of what the CPU should be executing. In this exercise, we will learn how to create two simple assembly files.

Assembly files are created using lines of the following syntax:

```
[label]:   <directive or opcode> [operands]
```

Directives are instructions for the assembler. They are used to specify how to interpret the assembly files and how to create the binary (i.e., setting which syntax or architecture - describing start of sections - etc..). Some of the assembly directives that we will commonly use are as follows:

```
        .intel_syntax noprefix          # assembler syntax to use <directive>
                                    # set assembly language format to intel

        .text                           # linker section <directive>
                                    # let the linker know that what follows
                                    # are cpu instructions to be executed
                                    # -- opposed to values that represent data.
                                    # For historical reasons cpu instructions
                                    # are called "text"

        .global _start                  # linker symbol type <directive>
                                    # makes the symbol _start in this case
                                    # visible to the linker
                                    # The linker looks for a _start symbol
                                    # so that it knows the address of the
                                    # first instruction of our program
```

On the other side, instructions are simple operations that the CPUs are designed and optimized to execute efficiently. We can think of assembly instructions as the atomic units for any program. Note that any program we write with any language would eventually be translated to a sequence of assembly instructions.

For more information on how to create assembly files, check the following sections in the lecture notes: `Lecture 7 : Assembly Programming Introduction`, `Lecture 8 : Writing some simple assembly programs` and in the textbook `14.5.3. Creating and using our first executable.`

Creating `popcnt.s`

We will create a simple assembly file that has the `popcnt` instruction inside. This instruction counts the number of bits set to 1 in the second operand (source) and returns the count in the first operand (a destination register). For example, if the value in rbx is 00010110, the instruction `popcnt rax, rbx` will write 3 to rax.

> **Note**
>
> `popcnt` is the mnemonic for the Intel population count instruction. The definition of population count of a binary vector is the number of 1's it contains. You can find more information about this Intel instruction in the Intel manuals.

- First, make sure that you are working in the cloned repository directory (check using `pwd` and `ls`). Then, create a new file called `popcnt.s`. Use your favorite editor or command for that. Eg. To do this with emacs, open a new terminal window, cd to the repository directory and run `emacs popcnt.s`.
- Remember that any file we create is just a file. When we add content to it, then it starts to have a purpose. The tool we will use, the GNU assembler, to process this file, will read the file contents and translate it into a fragment of a binary executable. To do this translation, the assembler scans the file for assembly directives, and then uses the directives to generate binary contents. As part of this processing, it can recognize the mnemonics for the CPU instructions and translate them into their binary opcodes (the binary numbers that the CPU recognizes). So the first thing we want to tell the assembler is what "syntax" of assembly code we want to use, which in our case is `Intel`. To do this, our first line should be the following:

```
.intel_syntax noprefix
```

- Let's continue building our assembly file. Our goal is to place two instructions into memory at a location that is associated with the label `_start`. To do this, let's make our file look like this.

```
        .intel_syntax noprefix
        .text
        .global _start
_start:
```

> **Note**
>
> Cool, at this point we are ready to place an instruction into memory that will be at the location of the label _start. We will learn more about labels as time goes on.

- Now, let's write a useful assembly instruction. Let's write the following:

```
    popcnt rax, rbx
```

> **Note**
>
> While the instructions we put in our assembly file are human readable strings such as `popcnt`, `add`, or `mov`, those instructions are eventually converted to binary code: just zeros and ones. Each instruction, in addition to its operands, is translated to binary code. In the above example, `popcnt rax, rbx` is the same as `.byte 0xF3, 0x48, 0x0F, 0xB8, 0xD8` written in binary code.

- To help us understand what the assembler is doing when it sees a mnemonic/instruction like `popcnt`, let's write the same instruction into memory again, but this time manually specifying the equivalent binary operation code.

```
    .byte 0xF3, 0x48, 0x0F, 0xB8, 0xD8
```

> **Note**
>
> When we use gdb to explore a process created from our binary, what do we expect to see at the memory location of `_start`? Think about what our two lines of assembly code are saying. Are they the same? Keep an eye out for what we see later.

- As discussed in class, it is nice to make sure that the CPU does not accidentally try to execute more instructions than we want. One way to ensure this is to place an instruction that requests the OS to stop and give control back to the debugger (in our case `gdb`). This will let us take control of the process again. To do this, we will finish our file with the following line.

```
    int3
```

Your file should look like this:

```
    .intel_syntax noprefix
    .text
    .global _start
_start:
    popcnt rax, rbx
    .byte 0xF3, 0x48, 0x0F, 0xB8, 0xD8
    int3
```

Don't forget to save your file. To do this, you can use the menu or easily press `c-x c-s` in emacs.

Creating `add.s`

Before we go on to review how we convert `popcnt.s` into an executable, let's practice what we have done so far by writing another similar assembly program.

We will create a second assembly file `add.s`. Instead of using `popcnt`, we will use `add`. This is the Intel instruction for integer addition.

We will create a simple assembly file to read two numbers stored in the memory together and then move the result back inside memory.

- Similar to the previous assembly file, create a new file called `add.s` using your favorite way and add to it the following line to specify the syntax:

```
.intel_syntax noprefix
```

- First, let's reserve some memory for placing our inputs and outputs. The `.data` section directive tells the assembler that we are now going to describe the data section.

```
    .data
```

- Our way to add a place for variables inside the memory is simply labeling the memory address for each variable start and then give the size and value. Add the following snippet inside your `add.s` file.

```
        .data
x:       .quad 142
y:       .quad 4200
sum:     .quad
```

- Now, we are ready to start writing useful instructions, let's add our instructions section.

```
        .text
        .global _start
_start:
```

- We need to execute three instructions for our task in this file as follows (modify the file to reflect it):

```
_start:
        mov rax, QWORD PTR x
        add rax, QWORD PTR y
        mov QWORD PTR sum, rax
        int3
```

> **Note**
>
> There are two distinct instructions we are using above. The first one is the `mov` instruction and the second is the `add` instruction. The `mov` instruction takes two operands; it moves data from the second operand to the first operand. Therefore, in `mov rax, QWORD PTR x`, we are moving data from memory to registers. In `mov QWORD PTR sum, rax`, we are moving data from registers to memory. The `add` instruction adds the value of the second operand to the first operand and stores the result in the first operand. More specifically, the instruction `add rax, QWORD PTR y` fetches the value of y from the memory, adds it to the value of register `$rax` and store the result in the register `$rax`.

Exercise 2: Creating Makefile

As we know, `make` is a cool tool that allows us to combine bash commands together. We most often use them for building executable files from source code files.

In the previous exercise, we created two assembly files. In this exercise, we will be automating the compilation of these files. Let's learn how to use `make` to do this. The goal is to be able to simply run the `make` command in the directory where our code is, and all of the commands that are needed to build the executable files will be automatically ran, as needed.

To tell `make` what to do, we place a file called `Makefile` in the same directory as our source code. In this file, we write "recipes" of commands.

So what are recipes? Recipes are a series of commands we want to execute when any of the source code files are modified – e.g., we update the code. This will save us the work of constantly having to run many commands when we make changes to our source code.

Let's work our way to a `Makefile` that will be able to automatically direct `make` to run the commands that:

1. convert the `popcnt.s` file into an executable called `popcnt`
2. convert the `add.s` into an executable called `add`.

Getting Started with `make`

First let's learn a little bit about how `make` behaves.

- Make sure you are inside your discussion repository for today and create a new file called `Makefile`. Again, you can do this in several ways but let's use our editor. Eg., if you have `emacs` running, use the menu and create a new file called `Makefile` or start `emacs` in a new terminal window with `emacs Makefile`.
- To get a sense for what make does, let's create our first recipe that echoes "Hello World!" to the terminal when we run `make hello`. To do this, add the following to your `Makefile`. **Be careful you MUST use a tab to indent the `echo` command.**

```
hello:
    echo "Hello World!" > hello
```

Note that recipes are structured in the following general format:

```
target: dependency_1 dependency_1
    recipe_command_1
    recipe_command_2
    recipe_command_3
```

> **Note**
>
> The target_name identifies a "recipe" of commands to run when the files listed as dependencies have changed. If no dependencies are given, then the recipe will always be run. Usually, the target is a file that the recipe commands will create. In our case, we don't have a dependency so the recipe command will always be run.

- Now create another recipe to say "Bye World!".

```
bye:
    echo "Bye World!" > bye
```

- Now, try the following commands, what do you see? **Don't forget to save your file. To do this you can use the menu or easily press `c-x c-s` in emacs.**

```
make hello
make bye
```

```
 - Try running `ls`.  What do you see?  Does this make sense?
 - Use `cat` to see what's inside the new files
```

> **Note**
>
> If you run `make` without specifying a target, it will run the first target in the `Makefile`. Eg., In our case, if you run `make`, it will run the `hello`.

Adding Recipes for ("popcnt.s" and "add.s")

Now let's rewrite our Makefile to add targets for creating our `popcnt` and `add` executables.

To create an executable from assembly source files, we need to run two commands:

1. Assemble: `as <source>.s -o <object>.o` eg. `as popcnt.s -o popcnt.o`
2. Link: `ld <object>.o -o executable` eg. `ld popcnt.o -o popcnt`

The following is an example of the recipes for creating the `popcnt` executable:

```
popcnt: popcnt.o
    ld popcnt.o -o popcnt

popcnt.o: popcnt.s
    as popcnt.s -o popcnt.o
```

Modify your Makefile using the above so that when you run `make popcnt` your binary executable is created. Use `ls` to verify. You can also run the resulting executable, but it might not be that interesting ;-)

Now add a target for `add` with the appropriate recipe.

Exercise 3: Using `gdb` with the Resulting Binary File

Now that we know how to create some binaries, let's use gdb to play with them.

1. Open a new terminal and start gdb on one of the binaries.
    - eg. `gdb -x setup.gdb popcnt`
2. Use the `gdb break` command to set a break point on the first instruction of your program.
    - `b _start`
3. Start a process from the binary.
    - `run`
4. At this point, the break point should hand control back to you before the first instruction.
5. Now let's poke around:
    - examine the bytes of your instructions
        - `x /12xb _start` – dump 12 bytes in hex format starting at the address of `_start`
        - `x /3i _start` – now ask gdb to `disassemble` 3 instructions worth of bytes starting at `_start`.
        - (How does it look like in the `popcnt` process?).
    - set some values into the registers (using set) and then use `stepi` to execute the instruction
    - observe what happens
    - set the pc back to the address of the first instruction `set $pc = _start`

Once you have played around with `popcnt`, try playing with `add`. Explore using your program as a glorified `calculator`.

> **Note**
>
> When exploring the `popcnt` binary, be sure to explore how the first 5 bytes (`x /5xb _start`) look, which we encoded using the line `popcnt rax, rbx`, and how the second 5 bytes (`x /5xb (_start+5)`) look, that we specified with with `.byte 0xF3, 0x48, 0x0F, 0xB8, 0xD8`. Also, use the the following to have gdb disassemble the 5 bytes separately: `x/1i _start` and `x/1i (_start+5)` or together `x/2i _start` with.

Final Note

This lab should help you with the practical steps for exploring simple assembly programs. Please see some suggested exercises in the lecture notes for things to try.

By Jonathan Appavoo