Assembly : GDB Intro

In this lab, you will be doing exercises to help get you started on:

1. Starting GDB
2. The basics of using GDB to control a process
    - set a break point
    - start a process from the binary
    - print the CPU registers of the process
    - examine memory of the process
    - single step instructions
    - set additional break points
    - continue execution
    - kill the process

The goal is to get us familiar with the mechanics. Later, we will focus on connecting the mechanics to what exactly is going on.

Setup

Create your github classroom lab repository by following the github classroom discussion link found under resources on Piazza.

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

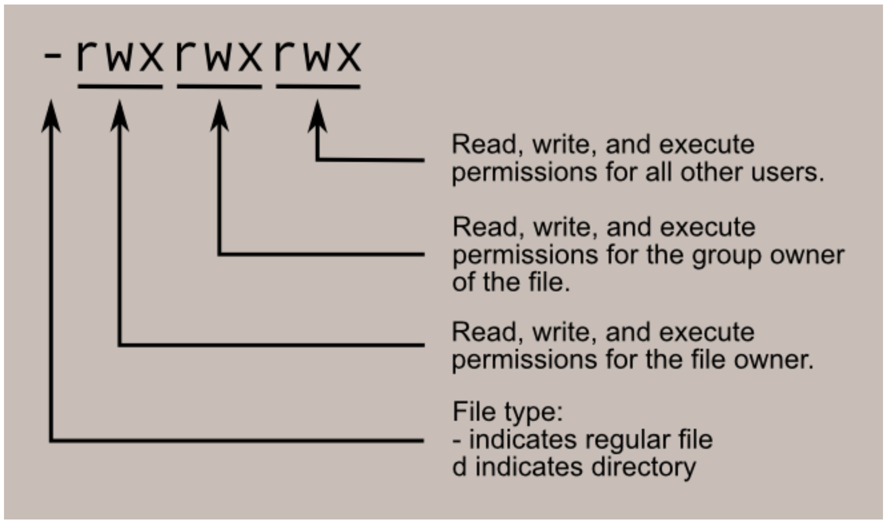
Exercise 1: What is GDB and How to Start It

> **What is GDB?**
>
> "GDB, the GNU Project debugger, allows you to see what is going on `inside' another program while it executes – or what another program was doing at the moment it crashed.
>
> GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:
>
> - Start your program, specifying anything that might affect its behavior.
> - Make your program stop on specified conditions.
> - Examine what has happened, when your program has stopped.
> - Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.
>
> Those programs might be executing on the same machine as GDB (native), on another machine (remote), or on a simulator. GDB can run on most popular UNIX and Microsoft Windows variants, as well as on Mac OS X."
>
> https://www.sourceware.org/gdb/

Our job in this exercise is to learn how to start gdb and have it start a new process from a binary, a process that we can then examine and control.

The binary that we will use in this lab is the binary for the UNIX `date` command. Its full path is `/usr/bin/date`. Take a moment and use `ls -l /usr/bin/date` and `file /usr/bin/date` to see the meta data for the date binary and what the `file` command has to say about it.

Remember you can interpret the seen permissions as shown in the figure below.

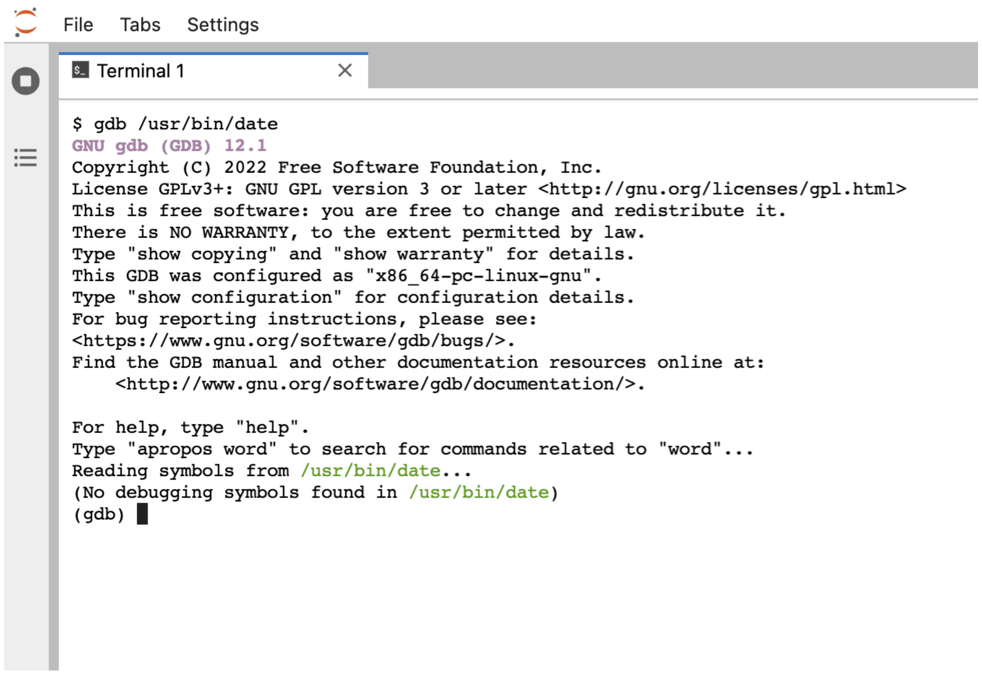

-rwxrwxrwx
Read, write, and execute permissions for all other users.
Read, write, and execute permissions for the group owner of the file.
Read, write, and execute permissions for the file owner.
File type:
- indicates regular file
d indicates directory

Starting and Exiting GDB

Starting

This step is pretty easy. We give the `gdb` an address to our executable.

```
gdb /usr/bin/date
```



```
$ gdb /usr/bin/date
GNU gdb (GDB) 12.1
Copyright (C) 2022 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-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://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 /usr/bin/date...
(No debugging symbols found in /usr/bin/date)
(gdb) 
```

At this point in this terminal, we are working with `gdb`, NOT the shell anymore. The commands we type will go to `gdb` and its responses will be displayed. This means that we must use gdb commands, not shell commands to do useful things. Before we exit, let's try running the `gdb` help command.

```
help
```

Take a moment and read.

Exiting

Great, now that we know how to start `gdb`, let's make sure we know how to exit.

The command to exit is:

```
quit
```

At this point, we should be back to the shell prompt.

Starting with a Setup File

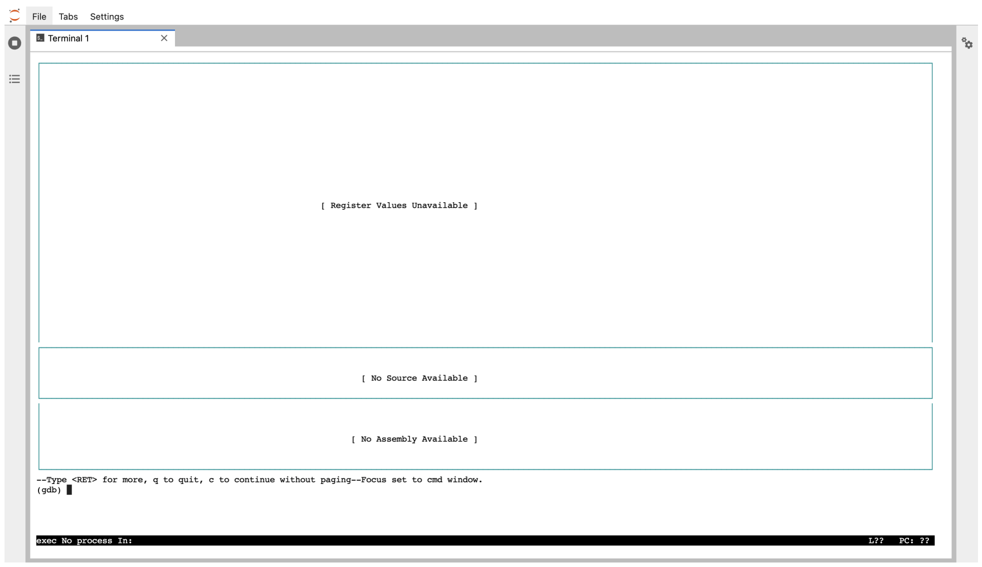
By default, gdb will start in its default line oriented mode. GDB also supports a more friendly "text user interface" (tui) mode. In this mode, it splits the terminal into different sub-areas that allow us to have it display various information to us while we issue gdb commands. To make it easier to start gdb in that mode and set it to a layout we want, we can use a setup file that contains the necessary gdb command. In our case, we have called this file `setup.gdb` and have placed the necessary gdb commands in it to setup gdb the way we would like. Feel free to see what is inside this file.

> **setup.gdb**
>
> We have provided the `setup.gdb` in the repository for this discussion. To use it, you will have had to cloned the discussion repository and your working directory will have to be the directory of the repository (eg., you will have to `cd` into the repository directory). For future assignments, you are free and encouraged to copy this file into your personal repositories.

Let's start `gdb` again but using this setup file. We can use the `-x` argument to tell `gdb` to execute the commands in our `setup.gdb` file automatically after it starts so that things are setup the way we would like.

```
gdb -x setup.gdb /usr/bin/date
```

Press the `ret` key and at this point your terminal should look something like this.



```
[ Register Values Unavailable ]

[ No Source Available ]

[ No Assembly Available ]

--Type <RET> for more, q to quit, c to continue without paging--Focus set to cmd window.
(gdb) 

exec No process In:                                                    L??   PC: ??
```

TUI mode

Our setup file directs GDB TUI mode to split the screen into four areas (top to bottom):

1. Register display
2. Source display
3. Disassembly display
4. Command area

And at the very bottom a status line is shown.

The Command Area is where we will issue gdb commands to control and explore a process that we start from the binary file we started gdb with. In our case, this binary file is `/usr/bin/date`. The other areas will be used by gdb to display information to us once we start a process running.

> **Note**
>
> In TUI mode, the terminal might get “messed up” sometimes. If so, try pressing Control and the l key (ctl-l). This should cause gdb to cleanup the display.

Starting a Process – Running the Program

So far, what we have done is just tell `gdb` that we will work with the binary file in `/usr/bin/date` using some settings. Remember that the binary file consists of memory and some instructions used to maybe modify the memory. Now, let’s try running the executable. The command to create and run a process from the binary is `run`. Try it out.

```
run
```

Since we did not set any “breakpoints”, `gdb` both started a process and ran the binary. In our case, the `date` program ran to completion and its output – the current time and date were displayed in the command area. `gdb` then printed its prompt indicating that it is ready for us to issue more gdb commands.

> **Note**
>
> The `run` command has options that let us pass arguments to the program just as if we had run a program as part of a shell command line. To learn more about the `run` command, you can use the `help` command eg., `help run`.

Exercise 2: Basics of Controlling a Process

Stopping Execution with a Break Point

In order to stop the process from running, we need to set a “break point” before issuing the `run` command. As we learn more about the CPU and process execution, we will understand how to know where to set a break point. For now, we will give you the command that will ensure that the CPU is stopped before it can execute the very first instruction.

```
break *0x00007ffff7fd0100
```

Note that most gdb commands have a long version of the command name and a short one. For the break command, we could have simply used `b` followed by the address where we need to insert a break point or any other flags.

> **Note**
>
> The above command instructs gdb to set a “break point” that causes the CPU to stop when the Program Counter’s value is 0x00007ffff7fd0100. This means that if the CPU attempts to fetch an instruction from the specified address, the process will be stopped and we will be able to use gdb commands to explore and control the process. The address we have specified is the first address of where the process created for the date command will execute an instruction from. In this way, we will get control once the process has been created but before even the first instruction has executed. Right now this might not make full sense but as we learn more, it will.

Since we have inserted a breakpoint, now if we enter the `run` command, instead of running the date program till completion, the date program process will stop and control will be handed back to us. Then, we can start poking around in `gdb`. You should notice that the display will now have information displayed in all but the source portion. Given that we are working with a “raw” binary, there is no source code – this is normal.

Registers

After running the process using `run` inside `gdb`, we can see the current values of the CPU Registers in the register display. We will be learning all about the CPU and its registers. At this point, it is sufficient to note that each has a name and a value. By default, the value of most registers are displayed in both hex and signed decimal notation.

We can also issue commands to display a specific register in the command area. To do this, we can use the `print` command with the register name prefixed with `$`. For example, to print the value of the Intel RAX register:

`print $rax`

To save some typing, `print` can be shortened to `p`. By default, the value of the register will be printed in signed decimal notation. The two most common notations we will use are hex and binary. To print in these notation, we would use `/x` and `/t`, respectively.

Eg.

```
print /x $rax
p /x $rax
print /t $rax
p /t $rax
```

Compare the values you got with the values in the register display part. Now, try printing out the values of `RSP` and `RIP` in both hex and binary. Ask for help if you get stuck.

> **Note**
>
> Depending on the size of your terminal, you might not be able to display all the registers in the register display at once. You can scroll up and down in each of the display areas by switching focus to the display area and then using the arrow keys to move up and down. To switch focus, use `ctl-x o`, that is press the control key and the x key together and release them, then press and release the `o` key. This will switch from one display area to the next. Remember you must make the command area the one in focus to issue commands.

We can also set the value of registers. Let’s try checking how the setting commands work by going through the help provided by `gdb`. To check the documentation, enter `help set` inside `gdb`. Skim through the main description.

Now, let’s try setting our first register `$rax`. Keep an eye on the values of the registers in the registers display part while we are doing changes.

```
set $rax = 0b00000001
set $rsp = 0b00000101
set $rip = 0b00000111
```

You should notice that the value for the registers `$rax`, `$rsp`, and `$rip` has been updated in both its displayed hex and binary forms.

Memory

Similar to registers, Memory can also be read and written at specific addresses. However, there are major differences though between what a register is and what a memory location is.

To read a memory location, we need to know a location and a size of the read. Memory locations are specified using addresses similar to those used to describe instruction addresses (i.e. `0x00007ffff7fd0100`). As for the read size, it is specified using two pieces of information. The first one is the data type and the second is how many elements of that data type we want to read. Note that there are various data types, with each having a different memory size (`unsigned char`, `unsigned int`, `unsigned long long`, …). To read from memory, we use the command `x`.

The following command reads from memory the first byte starting at address `0x00007ffff7fd0100`

```
x /1bt 0x00007ffff7fd0100
```

If we want to make it four bytes for example, we should change the number `1` before `bt` to make it `4`.

```
x /4bt 0x00007ffff7fd0100
```

Also, we can use the same command to read to other data types such as data types that require memory storage of 2 bytes, 4 bytes, and 8 bytes. (Check the outputs and count the number of digits in the outputed binary numbers.)

```
x /1ht 0x00007ffff7fd0100
x /1wt 0x00007ffff7fd0100
x /1gt 0x00007ffff7fd0100
```

Good, now since we know how we read data from memory, let's check how to write them. For writing, we use the command `set`. The following command line shows how we could set a memory for data type of char. Similar variations to that for `x` could be done here.

```
set {byte}0x00007ffff7fd0100 = 0b00000001
```

However, the executable we are using here is not compiled to allow for changing instructions in memory so we will get an error in this case.

Before continuing, let's reset the values we have just changed by re-runing the binary again. Use the `gdb` command `run` for this to happen. In the prompt, choose yes.

Execution

As you can see in the third display, we have a list of addresses where next to each address, we have instructions stored in those addresses waiting for execution in the right time. So how do we decide what the right time is for each instruction? Check the value of a special register called `$rip`. This register has the address of the next to-be-executed instruction. Each time an instruction is executed, the counter is increased to point to the next instruction. Sometimes, the value of this register `$rip` will be set to a totally different address than the next one, in cases of, for example, a branch instruction or also if we simply have a magic tool like `gdb` that can affect the value of registers.

First, let's verify that the register address `$rip` is acting as expected. Remember that currently, the `gdb` is just waiting for our control before executing the very first instruction since we have inserted a break point from the previous step. To tell `gdb` to execute the next command, we use a command called `stepi`. To learn more about this command, use `help stepi`.

```
stepi
stepi
```

The above commands should be enough for the program to execute two instructions and we see that the `$rip` register changes twice.

Now, let's say that I want to execute some instruction out of order. In other words, let's say I want to execute the first instruction again. We can simply do that using the `set` command and reissuing the `stepi` command. Eg.

```
set $rip = 0x00007ffff7fd0100
stepi
```

Note

While changing the `$rip` register to point to the first instruction will make the first instruction executed again, it may not give the same behavior as before; that is the case if the executed instructions have already changed some register values or memory. Remember that how your process behaves depends on the instructions stored in addition to the value of the memory or the mentioned registers.

Using `stepi` to advance one instruction at at time could be slow in some instances. Remember that a single program could have thousands of instructions, if not millions. Another way to navigate the program while executing is by adding another break point. Let's put another break point and then tell `gdb` to skip till it finds the next break point; for this we use a command called `continue`. Note that which break point the process will be paused at next depends on the order of execution, not the order of adding break points.

```
break *0x00007ffff7fd0df5
continue
```

Great! Now, that we have managed to go through the program and we explored its memory and instructions, we could simply kill this process by giving the command `kill`. Note that we could have also made the program continue execution normally by entering `continue` if we know that we will not hit any next break point.

```
kill
```

Additional information

The discussion describes very basic information about how to use gdb. Check chapter 17, specifically 17.2.1.2, 17.2.2.6, 17.3.1 and 17.3.2 in the textbook for more in-depth examples.

- gdb and registers Don't forget to click "click to show".
- gdb and memory Don't forget to click "click to show".
- gdb and instructions Don't forget to click "click to show".
- gdb and data Don't forget to click "click to show".