CS382 Computer Organization and Architecture — Fall 2021

Lab 8 · Debugging Assembly Programs Using `gdb`

Lecturer: Philippos Mordohai, Shudong Hao — *Date: October 26*

Contents

1 Objective — **1**

2 Task — **1**

Appendix A Installation — **2**

Appendix B Start Debugging — **3**

Appendix C Debugging Commands — **4**
- C.1 Breakpoints . . . 4
- C.2 Steps . . . 4
- C.3 Panel Focus . . . 5

Appendix D Printing Memory — **5**

Appendix E Inspecting Condition Codes — **6**

Appendix F Troubleshooting — **6**

1 Objective

In this lab, we are going to get familiar with GNU debugger (`gdb`), which allows us to set breakpoints and then inspect the internal states of a program.

`gdb` can be used to debug C program, which is the main topic of this lab. After writing assembly, we can also use `gdb` to debug assembly programs, and it'll be the content of another lab in the future.

2 Task

You have two tasks in this lab. First of all you need to install `gdb-multiarch` on your virtual machine. While the installation is simple, you would need to read Appendix carefully to start using it.

The second task is to find out your unique six character string. You're provided with an executable file `secret`. It asks for your Stevens ID, and produces a string based on the ID. The string is stored somewhere in the memory and is not printed out, so you have to disassemble the program, and use `gdb` to step through it to find out what your string is.

To run the program `secret`, you should use the `QEMU` command:

```
qemu-aarch64 -L /usr/aarch64-linux-gnu/ secret
```

This is fine if you're just curious. You can also generate a listing file using the following command:

```
aarch64-linux-gnu-objdump secret -D > secret.lst
```

and you can see the assembly code as well as the machine code in the file `secret.lst`.

What to Submit

You should submit a PDF report, including two parts. In the first part, clearly state your Stevens ID as well as the string you found out. Then in the second part, provide a detailed description on how you found out the string. It should include all the commands you used in `gdb`, screenshots, as well as your explanation.
Note: it is unacceptable to just explain the program and say what your string is. You have to show the steps in `gdb` to get points.

A Installation

The regular `gdb` we used in Lab 4 cannot be used here, because our program can only be executed in `QEMU` emulator, and the architecture is different. Therefore, we will have to install a `gdb` that can be used for multiple different architectures.

In your virtual environment, simply use the following command to install:

```
sudo apt-get install gdb-multiarch
```

We will use `gdb` to refer to `gdb-multiarch` in this lab.

B Start Debugging

We'll use `multiarch-gdb` and `qemu` together, so we need to open two terminals at the same time: one for `gdb` to step through, and the other for `qemu` to provide an emulated environment.

Note: for this lab, the assembled object file already has debugging information. If you want to use `gdb` on your own assembly file, you need to add `-g` flag with the assembler.

Assume the program binary is `a.out`. We first start `QEMU`:

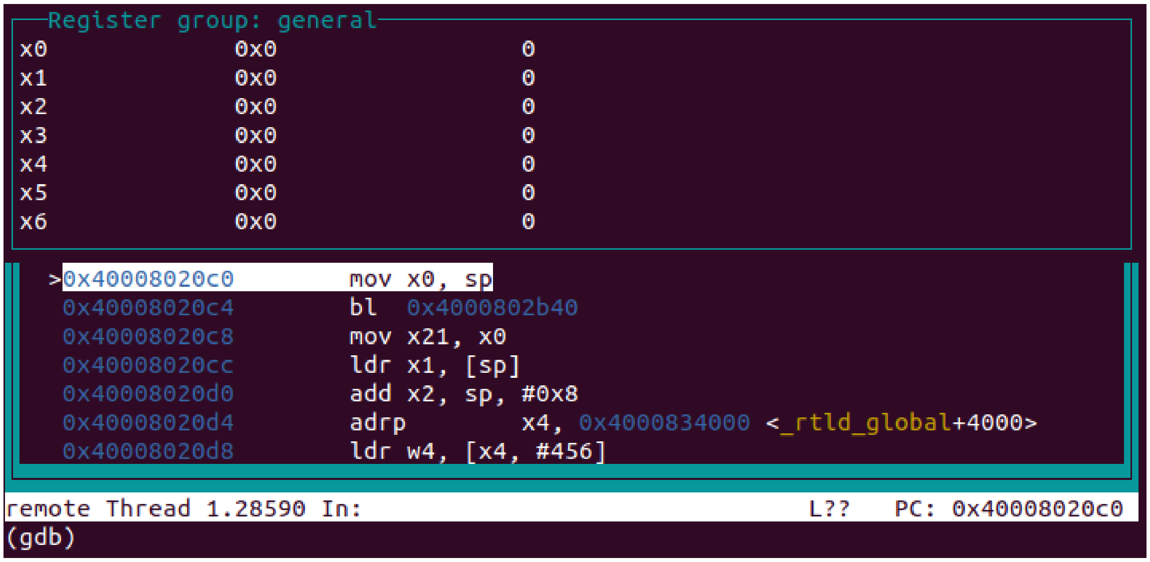
```
qemu-aarch64 -L /usr/aarch64-linux-gnu/ -g 1234 a.out
```

where the number `1234` is arbitrary, and it's a port for `gdb` to connect. Once this command is in, it'll freeze on the terminal and wait for us to start `gdb`. Now we can go to the other terminal and start `gdb`:

```
gdb-multiarch --nh -q a.out \
    -ex 'set disassemble-next-line on' \
    -ex 'target remote :1234' \
    -ex 'set solib-search-path /usr/aarch64-linux-gnu-lib/' \
    -ex 'layout regs'
```

Note there's a space between `remote` and `:1234`. The flags `-ex` are the commands we want `gdb` to execute in the beginning. If you don't add these flags when invoking `gdb`, you'll have to type them once you're in the `gdb` environment.

The interface you see is called TUI (Text User Interface).



C Debugging Commands

`QEMU` doesn't use a typical `run` command to start a program. You can think when you use command `qemu-aarch64` to invoke the program it already started and paused at the first instruction. So simply use command `continue` to start.

C.1 Breakpoints

When started `gdb`, the program is paused at somewhere in the static library. You can use the following command to set a breakpoint at the beginning of our program:

```
b _start
```

Then you can use `continue` command (or simply `c`) to reach our entry point. Other labels can be set as breakpoint as usual.

Note: if after you set the breakpoint to `_start` and used `continue`, but notice the break point is not exactly at label `_start`, see Troubleshooting section at the end of the document.

C.2 Steps

Most of the commands are pretty much the same to what you've been using for debugging a C program in `gdb`.

As a reminder, when you want to go to a procedure, you would need to use `step` or `s`. If the procedure is from a library, such as `printf`, it's not a good idea to step into it, so you can use `next` or `n`.

C.3 Panel Focus

When you enter `gdb` with assembly code and register group laid out, the default focus is the assembly code. Thus, when you use up/down keys on your keyboard, or scroll up/down using your mouse, it only works on the assembly code panel. To change focus to register group to view all registers, you can use command `focus regs`. To change back to assembly code panel, simply use `focus asm`.

D Printing Memory

It is quite often that we need to examine the data stored in memory. The syntax is as follows:

```
x/<length><format><unit> address
```

The `length` parameter specifies how much data we want to print from `address`. It can be both positive and negative integers.

The `format` parameter tells `gdb` in what format do you want to see the data. For example, if you pass `x`, then it will print the data in hexadecimal. If you pass `d`, it will print in decimal.

The `unit` parameter specifies how to group the data and interpret it.

For the `format` and `unit` parameters, there are many options. Please refer to the `gdb` documentation on https://sourceware.org/gdb/onlinedocs/gdb/Memory.html#Memory as well as https://sourceware.org/gdb/onlinedocs/gdb/Output-Formats.html#Output-Formats.

The `address` is the starting address in memory. It can be a label, or a hexadecimal address, or a register:

```
x/3xb 0x54320 # Print 3 bytes in hexadecimal starting from address 0x54320;
x/2xb $sp     # Print 2 bytes in hexadecimal from the stack pointer;
x/2db $x10    # Print 2 bytes in decimal from the address stored in x10;
x/5cb &hello  # Print 5 bytes in character from the label hello;
x/s &hello    # Print the content in a string from label hello until '\0'
```

In the following example, we created a string `format_str` in the data section. See how different parameters display the same content:

```
(gdb) x/10xb &format_str
0x411000:       0x48    0x65    0x6c    0x6c    0x6f    0x20    0x57    0x6f
0x411008:       0x72    0x6c
(gdb) x/10xc &format_str
0x411000:       72 'H'  101 'e' 108 'l' 108 'l' 111 'o' 32 ' '  87 'W'  111 'o'

0x411008:       114 'r' 108 'l'
(gdb) x/s &format_str
0x411000:       "Hello World!"
(gdb) x/5xt &format_str
0x411000:       01001000        01100101        01101100        01101100
 01101111
```

E Inspecting Condition Codes

To see condition codes, you can observe `cpsr` field in the register group. CPSR stands for Current Program Status Register.

`CPSR` is a 32 bit register, where different flags or conditions take different bits. We only care about the highest four bits: N, Z, C, and V:

31	30	28	28	0 – 27
N	Z	C	V	Other flags

The value displayed in register group panel for `cpsr` is usually in hexadecimal and decimal, so it's not very obvious to see individual bits. You can use the following command to print out the value in binary format:

```
p/t $cpsr
```

where `p` stands for print, and `t` for binary (two).

F Troubleshooting

You might notice that even if you set breakpoint to `_start`, `gdb` actually set the breakpoint a few bytes later after the label `start`. [1]

To set a breakpoint at the first instruction correctly, you can use the command `info files` to find out the actual address of `_start`:

[1] This typically doesn't happen on M1-based macOS, so try to set a breakpoint at `_start` at first, and if it's not exactly at the label, then proceed to use the actual address to set the breakpoint.

```
1 Symbols from "/home/shudong/demo/a.out".
2 Remote serial target in gdb-specific protocol:
3 Debugging a target over a serial line.
4             While running this, GDB does not access memory from...
5 Local exec file:
6             `/home/shudong/demo/a.out', file type elf64-littleaarch64.
7             Entry point: 0x400204
8 --Type <RET> for more, q to quit, c to continue without paging--
```

Notice in the output above, on line 7, we have the address of entry point `0x400204`, which is location of the label `_start`. Then we can set the breakpoint there:

```
b *0x400204
```