CSD1100

Assembler - Debugging

Vadim Surov

Introduction

- Debugging tools can be used to see what a program is **doing wrong** or why does it **crash** and **in education**.
- The object file generated by compiler contains **symbol table** with information about identifiers (type, size, constness, etc…).
- To visualize how the code is executed we need to add debugging information to this table by using the **`-g -F dwarf`** flags in compilation.
- In this presentation we go over how to access registers and memory addresses during debugging, which is more commonly needed when working at assembly level.

gdb

- The GNU Debugger (gbd) is a portable debugger that runs on many Unix-like systems and works for many programming languages.
- Most gdb commands are similar to the ones used for debugging any other programming language.
- To install it, run in WSL:

```
$ sudo apt-get install -y gdb
```

- Execute: `$ gdb`
- Then you will see a welcome information ended with `(gdb)` line like on the next slide. Use `-q` to skip the welcome part.

```
$ gdb
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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".
(gdb)
```

Using gdb

- Let’s look at the basics over a simple program:

```
section .text
  global _start
_start:
  mov rax, 60
  mov rdi, 0
  syscall
```

Using gdb

- If we save this code in a file named example.asm, we can generate the executable with these commands:

```
$ nasm -f elf64 -g -F dwarf -o example.o example.asm
$ ld -o example example.o
```

- We can now start gdb with the program loaded:

```
$ gdb -q example
(gdb)
```

- We can use the `b` command to set a breakpoint. For now, let's set it at the `_start` symbol:

```
(gdb) b _start
Breakpoint 1 at 0x400080: file example.asm, line 4.
```

Using gdb

- Because the executable contains debug information, gdb can tell us in which file and line number the breakpoint was set.
- We can now run the program and it will stop at our breakpoint:

```
(gdb) run
Starting program: /home/gdb/example

Breakpoint 1, _start () at example.asm:4
4    mov rax, 60
```

Using gdb

- The program stops at the first executable line of our program. We can step line by line using the s command:

```
(gdb) s
5    mov rdi, 0
(gdb) s
6    syscall
(gdb) s
[Inferior 1 (process 11727) exited normally]
Use q to quit gdb.
```

- Note: **s** is “step-into” command. Use **n** (next) or **c** (continue) command to run till the next breakpoint if you need “step-over” a function call

Inspecting registers

- Writing assembly code, you will find yourself moving things in and out of registers very often. It is then natural that debugging a program we might want to see their contents.
- To see the contents of all registers we can use `info registers` or the abbreviation `i r`.

```
(gdb) i r
rax            0x0   0
rbx            0x0   0
rcx            0x0   0
rdx            0x0   0
...
```

Inspecting registers

- In many cases we probably only want to see a specific register.
- To do this we just need to add the register name to the command: `i r <register>`:
- The first column is the hexadecimal value (0x0) and the second is decimal (0)

Inspecting memory

- Let’s introduce some variables to our program:

```
section .data
exit_code  dq 0
sys_call   dq 60
section .text
  global _start
_start:
  mov rax, [sys_call]
  mov rdi, [exit_code]
  syscall
```

Inspecting memory

- If we stop `gdb` at `_start`, we can inspect the variables in the program:

```
(gdb) print (int) sys_call
$1 = 60
```

- Note that we need to cast the variable to the correct type `(int)` or we'll get an error.
- Another thing we can do is get the memory address `sys_call` refers to:

```
(gdb) info address sys_call
Symbol "sys_call" is at 0x402008 in a file
compiled without debugging
```

Inspecting memory

- We can also see the data at a memory address using an asterisk (*):

```
(gdb) print (int) *0x402008
$4 = 60
```

General Tips

- gdb has very good help files. Type "`help [commandname]`" while in gdb.
- if you change and recompile your program in another window, you don't need to restart gdb. Just type `run` again, and gdb will notice the changes and load the new copy of your program.
- pressing enter executes the last command again. This makes it easily to step through your program line by line.

```
; Use this code to practice debugging.
; nasm -f elf64 -g -F dwarf -o main1.o main1.asm
; ld -o main1 main1.o
; gdb main1
; (gdb) b _start
; (gdb) run
; (gdb) s
; (gdb) i r rax
; (gdb) i r
; (gdb) p (int) sys_call
; (gdb) q

section .data
sys_call dq 60

section .text
  global _start
_start:
  mov rax, 60
  mov rdi, [sys_call]
  syscall
```

References

1. GNU Debugger in Wikipedia
   https://en.wikipedia.org/wiki/GNU_Debugger
2. Debugging assembly with GDB
   https://ncona.com/2019/12/debugging-assembly-with-gdb/