Homework #4

Advanced Programming in the UNIX Environment

Due: Jun 6, 2022 Extended to Jun 13, 2022 (hard-deadline)

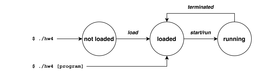
*** The difficulty of this homework has been simplified to handle only non-PIE programs.

Simplified Scriptable Instruction Level Debugger

In this homework, you have to implement a simple instruction-level debugger that allows a user to debug a program interactively at the assembly instruction level. You can implement the debugger by using the ptrace interface. The commands you have to implement are summarized as follows:

```
- break {instruction-address}: add a break point
- cont: continue execution
- delete {break-point-id}: remove a break point
- disasm addr: disassemble instructions in a file or a memory region
- dump addr: dump memory content
- exit: terminate the debugger
- get reg: get a single value from a register
- getregs: show registers
- help: show this message
- list: list break points
- load {path/to/a/program}: load a program
- run: run the program
- vmmap: show memory layout
- set reg val: get a single value to a register
- si: step into instruction
- start: start the program and stop at the first instruction
```

In a debugging process, you have to load a program first, configure the debugger, and start debugging by running the program. A debugger command may be only used in certain "states." The states include **any**, **not loaded**, **loaded**, and **running**. State **any** means that a command can be used at any time. State **not loaded** means that a command can only be used when a program is not loaded. State **loaded** means that a command can only be used when a program is loaded. State **running** means that a command can only be used when the program is running. The following is the state flow chart.



The details of each command are explained below. We use brackets right after a command to enclose the list of the state(s) that the command should support.

- **break** or **b** [**running**]: Setup a break point. If a program is loaded but is not running, you can simply display an error message. When a break point is hit, you have to output a message and indicate the corresponding address and instruction. The address of the break point should be within the range specified by the text segment in the ELF file and will not be the same as the entry point.
- **cont** or **c** [**running**]: continue the execution when a running program is stopped (suspended).
- **delete** [**running**]: remove a break point. Please remember to handle illegal situations, like deleting non-existing break points.
- **disasm** or **d** [**running**]: Disassemble instructions in a file or a memory region. The address of each instruction should be within the range specified by the text segment in the ELF file. You only have to dump 10 instructions for each command. If **disasm** command is executed without an address, you can simply output `** no addr is given.`. Please note that the output should not have the machine code `cc`. See the demonstration section for the sample output format.
- **dump** or **x** [**running**]: Dump memory content. You only have to dump 80 bytes from a given address. The output contains the addresses, the hex values, and printable ASCII characters. If **dump** command is executed without an address, you can simply output `** no addr is given.`. Please note that the output should include the machine code `cc` if there is a break point.
- **exit** or **q** [**any**]: Quit from the debugger. The program being debugged should be killed as well.
- **get** or **g** [**running**]: Get the value of a register. Register names are all in lowercase.
- **getregs** [**running**]: Get the value of all registers.
- **help** or **h** [**any**]: Show the help message.
- **list** or **l** [**any**]: List break points, which contains index numbers (for deletion) and addresses.
- **load** [not **loaded**]: Load a program into the debugger. When a program is loaded, you have to print out the address of entry point.
- **run** or **r** [**loaded** and **running**]: Run the program. If the program is already running, show a warning message and continue the execution. If the program is loaded, start the program and continue the execution.
- **vmmap** or **m** [**running**]: Show memory layout for a running program. If a program is not running, you can simply display an error message. The memory layout is:

  ```
  [address] [perms] [offset] [pathname]
  ```

  Check the demonstration section for the sample output format.
- **set** or **s** [**running**]: Set the value of a register
- **si** [**running**]: Run a single instruction, and step into function calls.
- **start** [**loaded**]: Start the program and stop at the first instruction.

Your program may output some debug messages. In that case, please add `****` prefixes before your message. We will remove lines beginning with `****` when comparing outputs.

Your program should read user commands from either user inputs (by default) or a predefined script (if `-s` option is given). Please check the demonstration section for the sample input and the corresponding output for more details about the implementation. The usage of this homework is:

```
usage: ./hw4 [-s script] [program]
```

Homework Submission

We will compile your homework by simply typing 'make' in your homework directory. Please ensure your Makefile works and the output executable name is correct before submitting your homework.

Please pack your C/C++/Assembly code and Makefile into a **zip** archive. The directory structure should follow the below illustration. The ***id*** is your student id. Please note that you don't need to enclose your id with the braces.

```
{id}_hw4.zip
└── {id}_hw4/
    ├── Makefile
    └── (any other c/c++/assembly files if needed)
```

You have to submit your homework via the E3 system. Scores will be graded based on the completeness of your implementation.

Demonstration

We use the hello world and the guess.nopie program introduced in the class to demonstrate the usage of the simple debugger. User typed commands are marked in ***blue***.

 # Load a program, show maps, and run the program (hello64)

```
$ ./hw4
sdb> load sample/hello64
** program 'sample/hello64' loaded. entry point 0x4000b0
sdb> start
** pid 16328
sdb> vmmap
0000000000400000-0000000000401000 r-x 0         /home/chuang/unix_prog/hw4_sdb/sample/hello64
0000000000600000-0000000000601000 rwx 0         /home/chuang/unix_prog/hw4_sdb/sample/hello64
00007ffe29604000-00007ffe29625000 rwx 0         [stack]
00007ffe29784000-00007ffe29787000 r-- 0         [vvar]
00007ffe29787000-00007ffe29789000 r-x 0         [vdso]
ffffffffff600000-ffffffffff601000 r-x 0         [vsyscall]
sdb> get rip
rip = 4194480 (0x4000b0)
sdb> run
** program sample/hello64 is already running
hello, world!
** child process 16328 terminiated normally (code 0)
sdb>
```

 # Start a prograrm, and show registers

```
$ ./hw4 sample/hello64
** program 'sample/hello64' loaded. entry point 0x4000b0
sdb> start
** pid 30433
sdb> getregs
RAX 0                   RBX 0                   RCX 0                   RDX 0
RSI 0                   RDI 0                   R8  0                   R9  0
R10 0                   R11 0                   R12 0                   R13 0
R14 0                   R15 0                   RBP 0                   RSP 7ffc51e88280
RIP 4000b0              FLAGS 0000000000000200
sdb>
```

 # Start a program, set a break point, step into instruction, continue the execution, and run the program again without start (hello64).

```
$ ./hw4 sample/hello64
** program 'sample/hello64' loaded. entry point 0x4000b0
sdb> start
** pid 74303
sdb> b 0x400055
sdb> b 0x4000ba
sdb> cont
** breakpoint @       400055: b8 01 00 00 00                  mov     eax, 1
sdb> si
** breakpoint @       4000ba: b9 d4 00 60 00                  mov     ecx, 0x6000d4
sdb> cont
hello, world!
** child process 74303 terminiated normally (code 0)
sdb> run
** pid 74325
** breakpoint @       400055: b8 01 00 00 00                  mov     eax, 1
sdb>
```

 # Start a program, set a break point, continue the execution, check assembly output, and dump memory (hello64)

```
$ ./hw4 sample/hello64
** program 'sample/hello64' loaded. entry point 0x4000b0
sdb> start
** pid 78354
sdb> disasm
** no addr is given.
sdb> disasm 0x400050
      400050: b8 04 00 00 00                  mov     eax, 4
      400055: bb 01 00 00 00                  mov     ebx, 1
      40005a: b9 d4 00 60 00                  mov     ecx, 0x6000d4
      40005f: ba 0e 00 00 00                  mov     edx, 0xe
      400064: cd 80                           int     0x80
      400066: b8 01 00 00 00                  mov     eax, 1
      40006b: bb 00 00 00 00                  mov     ebx, 0
      400070: cd 80                           int     0x80
      400072: c3                              ret
** the address is out of the range of the text segment
sdb> b 0x400064
sdb> disasm 0x4000c6
      4000c6: b8 01 00 00 00                  mov     eax, 1
      4000cb: bb 00 00 00 00                  mov     ebx, 0
      4000d0: cd 80                           int     0x80
      4000d2: c3                              ret
** the address is out of the range of the text segment
sdb> cont
hello, world!
** breakpoint @       4000c6: b8 01 00 00 00                  mov     eax, 1
sdb> disasm 0x4000c6
      4000c6: b8 01 00 00 00                  mov     eax, 1
      4000cb: bb 00 00 00 00                  mov     ebx, 0
      4000d0: cd 80                           int     0x80
      4000d2: c3                              ret
** the address is out of the range of the text segment
sdb> dump 0x4000c6
      4000c6: c6 01 00 00 bb 00 00 00 00 cd 80 c3 00 68 65  |.............he|
      4000d6: 6c 6c 6f 2c 20 77 6f 72 6c 64 21 0a 00 00 00 00  |llo, world!.....|
      4000e6: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
      4000f6: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00  |................|
      400106: 01 00 b8 00 40 00 00 00 00 00 00 00 00 00 00 00  |....@...........|
sdb>
```

 # Load a program, disassemble, set break points, run the program, and change the control flow (hello64).

```
$ ./hw4 sample/hello64
** program 'sample/hello64' loaded. entry point 0x4000b0
sdb> start
** pid 16690
sdb> disasm 0x400050
      400050: b8 04 00 00 00                  mov     eax, 4
      400055: bb 01 00 00 00                  mov     ebx, 1
      40005a: b9 d4 00 60 00                  mov     ecx, 0x6000d4
      40005f: ba 0e 00 00 00                  mov     edx, 0xe
      400064: cd 80                           int     0x80
      400066: b8 01 00 00 00                  mov     eax, 1
      40006b: bb 00 00 00 00                  mov     ebx, 0
      400070: cd 80                           int     0x80
      400072: c3                              ret
** the address is out of the range of the text segment
sdb> b 0x4000c6
sdb> l
  0:  4000c6
sdb> cont
hello, world!
** breakpoint @       4000c6: b8 01 00 00 00                  mov     eax, 1
sdb> set rip 0x400040
sdb> cont
hello, world!
** breakpoint @       4000c6: b8 01 00 00 00                  mov     eax, 1
sdb> delete 0
** breakpoint 0 deleted.
sdb> set rip 0x400040
sdb> cont
hello, world!
** child process 16690 terminiated normally (code 0)
sdb>
```

 # Load a program, set break points, run the program, and change the control flow (guess).

```
$ ./hw4 sample/guess.nopie
** program 'sample/guess' loaded. entry point 0x4006f0
sdb> start
** pid 17133
sdb> b 0x400879
sdb> cont
Show me the key: 1234
** breakpoint @ 5559c2a739cc: 48 39 d0                    cmp     rax, rdx
sdb> get rax
rax = 1234 (0x4d2)
sdb> get rdx
rdx = 17624781 (0x10ceecd)
sdb> set rax 5678
sdb> set rdx 5678
sdb> cont
Bingo!
** child process 17133 terminiated normally (code 0)
sdb>
```

Sample Scripts (30%)

 # Sample scripts passed to your homework (with -s option) can be found here!

Please note that the debugger is exited directly after the script is executed.

- hello1.txt (6%)
- hello2.txt (6%)
- hello3.txt (6%)
- hello4.txt (6%)
- guess.txt (6%)

Two examples of running scripts are given as follows.

#1. Print 'hello, world!' for three times.

```
$ ./hw4 -s scripts/hello3.txt 2>&1 | grep -v '^|*|*'
hello, world!
rip = 4194502 (0x4000c6)
hello, world!
rip = 4194502 (0x4000c6)
hello, world!
$
```

#2. Auto debugger for guess

```
$ ./hw4 -s scripts/guess.txt sample/guess.nopie 2>&1 | grep -v '^|*|*'
1234
rax = 1234 (0x4d2)
rdx = 580655839 (0x229c1adf)
Show me the key: Bingo!
$
```

Grading

- There will be seven advanced scripts used to test your homework, each worth 10 points.
- TAs will use the diff tool to compare the output of your program against the output of the `~/ta/hw4/ans_hw4` (except for messages prefixed with `****`). When comparing the outputs, continuous spaces and tabs in the output are merged into a single space character.
- You may run the executable from the TAs (`~/ta/hw4/ans_hw4`) and check its output for any unclear parts described in the spec. Your program output should output the same content to the sample executable except for debugging messages (messages prefixed with `****`) and randomized system memory addresses.
- We have provided a sample implementation with five sample scripts. Please access our online sandbox (tested only with Chrome, Firefox, and Edge; Safari is not supported). You can find everything in the directory `~/ta/hw4`. Note that you can only access the service within campus networks or via a valid VPN network.
  - You can use `-d` in `testcase_hello64.sh` and `testcase_guess.sh` to look at the diff result, and the result will save to `/tmp/testcase_hello{i}_diff` and `/tmp/testcase_guess_diff` respectively. However, we still suggest checking it carefully by yourself to avoid any mistakes ignored by the additional diff filter.

Hints

Here we provide some hints for implementing this homework.

- For disassembling, you have to link against the capstone library. You may refer to the official capstone C tutorial or the ptrace slide for the usage.