Introduction

Having completed these exercises, you should feel comfortable using the GNU debugger "gdb" to diagnose errors in C programs. You will also be able to use the debugger to step through assembly code and inspect the registers and memory locations of a running program.

A debugger is a software tool used to inspect the execution of a program. Using a debugger, you can run your application in an environment that controls how the code runs and gives you tools to poke around the memory, stack, and even assembly translation of the program. GDB is the GNU debugger, which we will be using in this lab. All the features we cover are available in any other modern debugger you may use later.

Do this lab on a CSIL Linux machine; that is, log into one of linux1, linux2, and so on up to linux7. The programs used, gdb and gcc, have different behavior on different platforms.

Walkthrough

These steps will work through the features of gdb that you will use most commonly to diagnose most problems in your code. Even if you are familiar with gdb from a previous course, please take the time to do this walkthrough. The information here will assume that you are following along in your terminal, typing all the commands that are indicated here, and taking the time to understand the output they generate, as explained by the commentary here.

Getting your program ready

The compiled binary output from gcc is just a pile of machine code. If you run gdb on a raw binary from gcc with no other information available, gdb will be able to turn the machine code into assembly but will not know the mapping from the assembly back to source code instructions. Run the following commands to launch a program without compiler debugger support, just to see what that looks like. Note that in the following, we will prefix commands to be entered at the unix shell with `$`.

```
$ cd ~/cs144/CNETID-cs144-win-25/
$ git pull
$ cd lab2/walkthrough
$ make
$ gdb walkthrough
(gdb) start
(gdb) where
```

gdb is usually either started with no arguments or just with the name of the program that you will execute. Once gdb is started, the prompt changes to `(gdb)`. The `start` command will run the program specified and stop execution at the first instruction of the `main` function. When the program is stopped, the `where` command prints out the current stack. In this case, it will look something like `0x080483e7 in main()`. Any time you see a hexadecimal address (code memory locations) in the output of the `where` command next to the function name, instead of file names and line numbers, it means the program you are looking at was not compiled with debugger support enabled.

Within gdb, use the `kill` command (followed by `y`) to stop running the program so that we can rebuild with debugging support. Leave gdb running in that terminal window and open up a second terminal window. Since gdb can tell when a program has been rebuilt, you can have a separate terminal window in which to build and write code.

In the second terminal window, `cd` into `lab2/walkthrough`, and edit `Makefile` (for example with `emacs Makefile`) to add debugging information to the program. To the `CFLAGS` variable definition, add the `-g` flag, which causes debugging information to be included in the compiled program. Remake:

```
$ make clean
$ make
```

Note that when you change compiler flags in a `Makefile` you will often need to `make clean; make` because `make` does not know that build targets depend on variables.

Back in the first terminal window, the one running gdb, enter the following commands:

```
(gdb) start
(gdb) where
```

In response to `start`, gdb should have printed something like `'walkthrough' has changed; re-reading symbols`, since gdb noticed that you re-compiled your code. And now, the `where` command will print out something like:

```
#0  main (argc=1, argv=0xffffdbc4) at main.c:11
```

That is more useful than the previous `where` output, particularly when using more advanced features of the debugger.

Breakpoints

A breakpoint is a way of telling the debugger to stop executing the statements of a program when a given location is reached. The locations can be source files with line numbers, function names, or literal code addresses. The following will set a few breakpoints in the test program:

```
(gdb) break compute
(gdb) break main.c:32
(gdb) break main.c:36
```

These commands will set three breakpoints: one at the entry point to the function compute, and one each on the lines that change the value of the local variable named result. Each breakpoint will be numbered by gdb so that you can issue commands to disable or temporarily skip a breakpoint by name. To see all the breakpoints currently set:

```
(gdb) info breakpoints
```

which should produce something like:

```
Num     Type           Disp Enb Address    What
3       breakpoint     keep y   0x0804842f in compute at main.c:23
4       breakpoint     keep y   0x08048446 in compute at main.c:32
5       breakpoint     keep y   0x08048452 in compute at main.c:36
```

Now continue execution of the program with:

```
(gdb) continue
```

The program should have stopped at the beginning of the `compute` function in `main.c`. gdb will print out the reason for stopping. In this case, it prints the number of the corresponding breakpoint. Use the `where` command again:

```
(gdb) where
#0  compute (i=10) at main.c:23
#1  0x08048409 in main (argc=1, argv=0xffffdbc4) at main.c:15
```

This is showing that at the current (stopped) point of execution, there are two functions on the call stack: `main`, which has called `compute`.

To no longer stop at this breakpoint, disable it with the following command (because we should be stopped at breakpoint 3):

```
(gdb) disable 3
```

To remove all breakpoints:

```
(gdb) delete breakpoints
```

(followed by `y`)

While `where` shows you the current call stack, the `list` command will show you the line of code currently being executed and five lines of context in each direction. You can provide different arguments to list to see different parts of source files (use `help list` for more examples).

```
(gdb) list
```

After hitting a breakpoint, you can continue the program again until another breakpoint is hit or the program terminates. Alternatively, there are commands for controlling execution with finer granularity. The following are the three you will use most often:

- `step` executes to the next line of source code (no matter how many expressions are in the current one!). It moves line by line through any part of your program. Note that the next line may be within a function you are calling on the current line; in this sense, you "step in" to function calls.
- `next` executes to the next source line in this function or to the end of this function. It is used when you are debugging an individual function and don't care about any calls to other functions. Note that you can think of this version as "stepping over" function calls. (To be clear, the function call still gets executed, but you will next see the state of the program after it returns.)
- `finish` executes to the end of this function. It is used when you are done looking at a function call and want to finish it off and return to the caller (or when you accidentally "step" when you mean to "next").

Try using `step` and `next` a few times, and then

```
(gdb) finish
```

Type `continue` when you are done to execute to the end of the program; this will produce something like

```
Continuing.
0
[Inferior 1 (process 10672) exited normally]
```

Note that you will remain in the debugger even after the program exits.

Inspecting data

Of course, just executing and breaking in the program to see where control flows isn't very useful. Instead of relying on `printf` statements and recompilation to show values, the debugger offers a powerful set of tools for inspecting variable, register, and memory values.

Following the steps above, you can start running the program again with:

```
(gdb) start
```

Or if you've quit `gdb` for some reason, you can restart with

```
$ gdb walkthrough
(gdb) start
```

Execution should now be stopped at the first line inside `main`, just before the variable initialization occurs. Type

```
(gdb) info locals
```

This will list all local variables and their values. If this were a C++ method, it would include the `this` pointer as well. Note that if any of the variables have a value of "`value temporarily unavailable, due to optimizations`" then you are running in an optimized build of the program (with `-O` passed to gcc). This likely means that the value is probably going to be stored in a register but has not been initialized yet. In that case, inspection of the assembly is necessary to understand what happened. More on that later.

Most interestingly, observe that the values of the variables `result` and `i` have not yet been initialized and contain trash data. This trash data may be zero or may be random, depending on the implementation of the C runtime and the state of memory. Do the following:

```
(gdb) step 2
(gdb) info locals
```

The `step` command can be given a number for the number of times to step, as done here. Now, execution is about to proceed into the call to the `compute` function and both of the local variables have been initialized.

To print out a specific value you're interested in, use the `print` command. Use the following commands to step into the `compute` function and view the value of the new local variable, which is conveniently also named `i`.

```
(gdb) step 3
(gdb) print i
```

If you'd like to see that value every time you step, so you don't have to keep typing `step` then `print i`), you can use `display` instead. `print` and `display` also take a simple C expression as well, so you can even perform some basic operations within the debugger:

```
(gdb) display i*2+1
(gdb) step
(gdb) step
(gdb) step
```

Note how after the third `step`, you were able to see the value of `i*2+1` change from 21 to 19. The `undisplay` command turns off any display expression you have set.

Watchpoints

Sometimes, breakpoints are tedious. For example, you might want to break every time a value changes. The traditional way to see every time a value changes with breakpoints is to set a breakpoint on every line that changes the value. Instead, you can set a watchpoint on that specific value, and the debugger will stop any time that it is written.

```
(gdb) watch result
(gdb) cont
```

The walkthrough will continue execution and run until the `result` value is updated. It will display both the previous and new values. Watchpoints are extremely helpful, as they also work on data in arrays and function appropriately in the presence of aliasing. So, if you have an array in memory, you can put a watchpoint on a value in that array, and no matter which pointer the program uses to write that value, the debugger will still break on the watchpoint for that memory location.

You can get a list of all watchpoints and disable watchpoints with the same command as breakpoints:

```
(gdb) info breakpoints
(gdb) disable 7
```

where 7 is the `Num` of the breakpoint (left-most column in the output of `info breakpoints`); the numbering might be different if you've restarted gdb during this walkthrough.

Disassembly

Sometimes (like for Project 1), you need to see the assembly representation of the machine code being executed. Usually, this situation occurs because either you don't have the source or debugging information for a program, but you still need to understand in detail the behavior of the program. To see the assembly representation of the machine code being run, use the `disassemble` command combined with a `print` of the program counter register:

```
(gdb) disassemble
(gdb) print/x $pc
(gdb) list
(gdb) where
```

By default, `disassemble` will show you the function you are currently in. You can also provide it with the name of another function and it will show you a dump of that one. To understand which assembly instruction you're on, use output from the `print` command above (which has been modified to "show hex value" via the `/x` flag; try `help print` and `help x` if you want to learn more). The `print` command on the program counter is used instead of the `where` command because gdb's `where` command will only print the instruction code address if execution is stopped at the first single instruction associated with a line of C code. The register `$eip` will also show the current instruction address. Be careful with syntax - while assembly uses the % character to refer to registers, gdb uses the $ character.

Your output from the above might look something like:

```
(gdb) disassemble
Dump of assembler code for function compute:
   0x08048429 <+0>:push   %ebp
   0x0804842a <+1>:mov    %esp,%ebp
   0x0804842c <+3>:sub    $0x10,%esp
   0x0804842f <+6>:movl   $0x0,-0x4(%ebp)
   0x08048436 <+13>:jmp    0x8048458 <compute+47>
   0x08048438 <+15>:subl   $0x1,0x8(%ebp)
   0x0804843c <+19>:mov    0x8(%ebp),%eax
   0x0804843f <+22>:and    $0x1,%eax
   0x08048442 <+25>:test   %eax,%eax
   0x08048444 <+27>:jne    0x8048452 <compute+41>
   0x08048446 <+29>:mov    -0x4(%ebp),%eax
   0x08048449 <+32>:imul   0x8(%ebp),%eax
   0x0804844d <+36>:mov    %eax,-0x4(%ebp)
   0x08048450 <+39>:jmp    0x8048458 <compute+47>
   0x08048452 <+41>:mov    0x8(%ebp),%eax
   0x08048455 <+44>:add    %eax,-0x4(%ebp)
=> 0x08048458 <+47>:cmpl   $0x0,0x8(%ebp)
   0x0804845c <+51>:jg     0x8048438 <compute+15>
   0x0804845e <+53>:mov    -0x4(%ebp),%eax
   0x08048461 <+56>:leave
   0x08048462 <+57>:ret
End of assembler dump.
(gdb) print/x $pc
$3 = 0x8048458
(gdb) list
21
22
23int compute(int i) {
24  int result = 0;
25
26  while (i > 0)
27  {
28    i--;
29
30    if (i % 2 == 0)
(gdb) where
#0  compute (i=9) at main.c:26
#1  0x08048409 in main (argc=1, argv=0xffffdbc4) at main.c:15
```

From the `print` command, you can see that the program counter (PC) is at `0x08048458`, which in our example `disassembly` output corresponds to "`0x08048458 <+47>:cmpl $0x0,0x8(%ebp)`" Judging from the output from `list` and `where`, we ended after the last watchpoint. The `result` variable has just been changed, and we're about to perform the comparison for the while loop's conditional, to decide whether to go through the loop body again or fall through to the return statement.

To follow the individual assembly instructions, there are "`i`" versions of `step` and `next`:

```
(gdb) stepi
(gdb) nexti
```

Besides using gdb to incrementally execute individual assembly instructions, you can also inspect the values of all the registers we've learned about so far, with `info registers`.

```
(gdb) info registers
eax            0x99
ecx            0xffffdbc4-9276
edx            0xffffdb54-9388
ebx            0xf7fb1ff4-134537228
esp            0xffffdae80xffffdae8
ebp            0xffffdaf80xffffdaf8
esi            0x00
edi            0x00
eip            0x80484380x8048438 <compute+15>
eflags         0x206[ PF IF ]
cs             0x2335
ss             0x2b43
ds             0x2b43
es             0x2b43
fs             0x00
gs             0x6399
```

Finally, to quit gdb, `kill` (terminate) the application and then quit.

```
(gdb) kill
(gdb) quit
```

There is nothing to `git add .; git commit -m "lab2"` for the walkthrough.

Exercise 1

The files needed for this exercise are in the `ex1` subdirectory. There are two `.c` files, `test.c` and `library.c`. Do not edit `test.c`: it is the driver program performing tests of the code in `library.c`. Run `make` to create the `test` program.

There are two tests that will run when you execute the binary `test`, and both of them will initially fail. You should set appropriate breakpoints in the `library.c` source to stop execution in the functions being tested (`multiStrlen` and `twoFingerSort`), inspect the arguments, and fix the functions to match the behavior expected by the test. Remember to re-run `make` after you make any changes to the source code!

Don't try to just return the right answer (i.e. 42 or NULL) to kludge the tests to pass - fix the code!

Exercise 2

Make and run the program in the `lab2/ex2` directory. Note that it's not printing out anything, even though `stringToPrint` has a value! Run it under the debugger. When and how is the array that gets printed getting corrupted? Specifically, on which line does it occur, and what are the values of the local variables at one point where a corrupted value is written? What debugger commands did you use? You do not need to fix this program.

Exercise 3

Doing this exercise is a good test of the skills you will need to complete Project 1.

There is a 32-bit x86 linux binary `bin-linux` in the `ex3` directory. There are no sources or debugging information available. `cd` in to the directory and run `chmod 700 bin-linux` to make it executable. Use gdb to load it, use the start command, and use the debugging commands from the walkthrough to answer the following questions.

1. How much space is being reserved for locals on the stack frame? Ignore the first three lines of the assembly - those instructions are administrative setup related to stack frame alignment.
2. After the stack is set up but before a call is made to the function, a pointer value is stored into a local. What is the string at the address location being pointed to?
3. Set a breakpoint on the assembly instruction after returning from the function call. What debugger command did you use to do that?
4. What value was returned from the function call? Remember what particular register holds the function return value, and how we can inspect the values of registers.

Make sure you know how to answer these questions! Having the skills to answer these questions is more important than simply knowing the answers. You can find the answers in `.answers.txt` (note the leading "`.`").

(Based on lab2 originally written for cs154 by [Lars Bergstrom])