← Lab 4: gdb

Finish by midnight, Wednesday 10/3

Debugging

Debugging is the process of figuring out *why* your program is broken. Not only do you fix your bugs, but you also get a deeper understanding about why your program is incorrect. Then you can avoid those mistakes in the future.

Oh, that's what happened.

A **debugger** is a tool to help with this process. If your program fails, a debugger lets you watch it fail step-by-step, so that you can figure out what went wrong, and when. I love this quote: "a debugger lets you watch your program crash in slow motion."

`printf` **debugging** is probably all you've used so far: you stick a bunch of prints into your code to print out the values of variables, or to say `"got here"`. You can get pretty far with this, but it's tedious.

What's `gdb`?

`gdb` is the GNU debugger. It's, uh, a little difficult to use, but really powerful. Like most things that the GNU people make.

How does a debugger work?

A debugger is a sort of "supervisor." It has full control of your program: it can pause, resume, run it step-by-step, look at all the variables, *change* all the variables, etc.

Probably the most important thing a debugger can do is **pause** with something called a **breakpoint.** A breakpoint is a way of telling the debugger, "when my program gets to this line, stop!"

Once the program is paused, you can look at everything, see where you are, see what went wrong, and so on. It's like stopping time.

---

1. Things are bad!

Login to thoth and `cd` into your private directory. Make a directory for this lab, and in there (Don't forget the period at the end!):

```
cp ~jfb42/public/cs449/gdbdemo.c .
```

Take a look at the contents of `gdbdemo.c` There are some mistakes in there. Don't fix them yet!

Compile it and run it to see the result:

```
(23) thoth $ gcc -o gdbdemo gdbdemo.c
gdbdemo.c: In function 'less_fun':
gdbdemo.c:31: warning: initialization makes pointer from integ

(24) thoth $ ./gdbdemo
Enter a number: 3
Segmentation fault (core dumped)
```

Unlike in a Java program, when our program crashes, we don't get a nice printout showing exactly where the error happened. Why?

When we compile an executable, most of the "squishy human" things like variable names, function names, references to the C source code etc. are thrown away. The CPU doesn't need or care about them, and they would just waste space in the executable file.

But we can tell `gcc` to include that information with the `-g` flag. So do that:

```
(25) thoth $ gcc -g -o gdbdemo gdbdemo.c
gdbdemo.c: In function 'less_fun':
gdbdemo.c:31: warning: initialization makes pointer from integ
```

---

2. Running it in `gdb`

Now we can run our program through `gdb`. Since it's a supervisor, we tell it what program to run:

```
(26) thoth $ gdb gdbdemo
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-64.el6_5.2)
```

```
...blah blah blah...
Reading symbols from /afs/pitt.edu/home/j/f/jfb42/public/cs449
(gdb) _
```

`gdb` is an interactive program with its own prompt where you type commands.

When you start gdb like this, it does not run your program. You have to run it with the `run` command (or just `r` for short).

When we run the program, it asks for a number again, and we type it, and...

```
(gdb) r
Starting program: /afs/pitt.edu/home/j/f/jfb42/public/cs449/gd
Enter a number: 3

Program received signal SIGSEGV, Segmentation fault.
0x00000034ff656f50 in _IO_vfscanf_internal () from /lib64/libc
Missing separate debuginfos, use: debuginfo-install glibc-2.12
(gdb) _
```

`gdb` automatically pauses our program right as it crashes, so we can see what went wrong. But where are we? What the heck is `_IO_vfscanf_internal`?

You can ignore the "Missing separate debuginfos" message.

Let's get a **stack trace.** This shows the list of activation records on the stack, and the line where that function is running (or waiting for another function to return).

Use the `where` or `bt` command:

```
(gdb) where
#0  0x00000034ff656f50 in _IO_vfscanf_internal () from /lib64/
#1  0x00000034ff664daa in __isoc99_vsscanf () from /lib64/libc
#2  0x00000034ff664d28 in __isoc99_sscanf () from /lib64/libc.
#3  0x0000000000400605 in main () at gdbdemo.c:11
```

We can see some kind of internal functions from `libc.so.6`, and we can ignore those. The important line here is:

```
#3  0x0000000000400605 in main () at gdbdemo.c:11
```

That says whatever is on line 11 of `gdbdemo.c` is what crashed. So let's have a look at that using the `list` command:

```
(gdb) list gdbdemo.c:11
6               int x;
7               char buf[30];
8
9               printf("Enter a number: ");
10              fgets(buf, sizeof(buf), stdin);
11              sscanf(buf, "%d", x);
12
13              fun();
14
15              return 0;
```

Wait a second. The `scanf` functions expect *addresses* for their arguments. We should have written `&x`.

Quit `gdb` with the `quit` or `q` command and say yes, exit.

Then you can fix that mistake, recompile, and run it again.

---

3. THINGS ARE STILL BAD OH NO

Now you can type in a number successfully, but you get another error, a SIGFPE:

```
Enter a number: 34

Program received signal SIGFPE, Arithmetic exception.
0x000000000040068f in fun () at gdbdemo.c:24
24              int c = a / b;
```

This time it even shows the line of code that crashed. A division! What kind of thing could happen during a division that causes an exception? (HMMMMMM)

We can print out all the local variables of `fun` with `info locals`:

```
(gdb) info locals
a = 5
b = 0
c = 52
```

Well well well. `b` is 0. We can't divide by 0 now can we?

Exit gdb, fix the code (change `b` to something else), recompile, and rerun.

---

4. why won't things get better

Now we're crashing on this line:

```
Program received signal SIGSEGV, Segmentation fault.
0x00000000004006e5 in less_fun () at gdbdemo.c:33
33                      printf("*q = %d\n", *q);
```

Let's print out the locals again.

```
(gdb) info locals
p = 0x601010
q = 0x2d
```

Notice – this pointer is definitely not NULL, but it crashes anyway, since it's definitely not valid.

Uh. Ok. That's weird. `q` doesn't look like a proper pointer value.

Let's watch `less_fun` crash in slow motion. We can do this by setting a **breakpoint** on it. This tells `gdb` "pause right before you this thing gets executed."

```
(gdb) break less_fun
Breakpoint 1 at 0x4006b6: file gdbdemo.c, line 29.
```

Now we can **restart** the program with the `r` (`run`) command again.

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /afs/pitt.edu/home/j/f/jfb42/public/cs449/gd
Enter a number: 3
Result is 5
```

```
Breakpoint 1, less_fun () at gdbdemo.c:29
29              int* p = malloc(sizeof(int));
```

Okay. It says we're on the `malloc` line, which means **that is the line about to be run.** It hasn't run yet.

We can use the `n` (`next`) command to run to the next couple lines.

```
(gdb) n
30              *p = 45;
(gdb) n
31              int* q = *p;
```

Okay. Now `p` has been assigned, and so has `*p`. Let's print those out with… `print`!

```
(gdb) print p
$1 = (int *) 0x601010
(gdb) print *p
$2 = 45
```

You can use any C expression with `print`. For example I could even write this:

```
(gdb) print p[0]
$3 = 45
```

Hey, it's the same thing as `*(p + 0)`, right?

Back on track. Let's use `n` to run the `int* q = *p;` line. Then, let's print `q`.

```
(gdb) n
32              if(q != NULL) {
(gdb) print q
$4 = (int *) 0x2d
```

Uh… okay. Wait. `0x2d`. What is that in decimal? We can use `print/d` to print it in decimal.

```
(gdb) print/d q
$6 = 45
```

Oh!!!!!!!! 45. 45 is the value we put in `*p`. Ohhhh that's what that compiler warning was about "makes pointer from integer" blah blah ohhhh okay. Okay. *Okay*.

Now you know what's wrong, and you can fix that bug. And there are no more!

Common commands

Now you can play with `gdb` some more. Try it on your previous labs, or debug your project 1 if you couldn't fix something!

You can learn more about all of the commands by typing `help`, or on a specific command by typing `help command_name` e.g. `help bt`

Command	Shortcut	Description
`help`		Get help on a command or topic
`apropos`		Search the help for a term
`set args`		Set command-line arguments
`run`	`r`	Run (or restart) a program
`quit`	`q`	Exit gdb
`break`	`b`	Place a breakpoint at a given location
`continue`	`c`	Continue running the program after pausing
`backtrace`	`bt`, `back`	Show the function call stack
`where`		Same as `backtrace`
`next`	`n`	Go to next line of source code (doesn't follow calls)
`step`	`s`	Go to next line of source code (follows calls)
`nexti`	`ni`	Go to the next instruction (doesn't follow calls)
`stepi`	`si`	Go to the next instruction (follows calls)
`print`	`p`	Print the value of an expression written in C notation
`x`		Examine the contents of a memory location (pointer)
`list`	`l`	List the source code of the program
`disassemble`	`disas`	List the machine code of the program

Important takeaways

- If your program crashes, your *first instinct* should be to run it in `gdb` to find out where it's crashing.
    - **DON'T GUESS WHERE YOUR PROGRAM IS CRASHING.**
    - Often it's not where you think.
- If you've checked a pointer for NULL, that's not necessarily a guarantee that it's valid.
- If you come to us without having debugged your programs…
    - we will tell you to debug it and come back if you're totally stuck.

What to turn in

Nothing! This lab is for practice only. Believe me, you'll need to use the debugger when working on your memory allocator.