Core Dump

Scripting GDB

Posted on February 1, 2014.

Motivation

Two semesters ago, I was introduced to gdb in my systems programming course. Since this was the first time that I used an actual debugger, I was really impressed by the things it could do. I wanted to see what gdb was capable of, so I tried every single command on simple programs. Unfortunately, by the time I actually needed gdb in my class projects I forgot most of them.

To be honest, I was just using gdb whenever I got segmentation faults from my programs. I would quickly run to the point where the program had the segmentation fault and print a bunch of variables to see what was wrong with it. Sometimes, I'd also do a backtrace. This is probably more than enough for small and simple programs but when complexity grows it barely helps debugging. On many occasions, the bug originates in a point in the code that is distant from the point in the program where the existence of the bug became apparent. In these cases, just skimming to the end where the error shows up, doesn't really help.

Being a tutor for CS classes in my school, I had a lot of students that had buggy programs which misbehaved this way. Since their programs had a large number of lines for course projects, I decided to encourage them to use gdb more often and learn more commands. Although that helped 30% of the students, the majority of them still had problems. They didn't want to use gdb because it was really tedious and repetitive for them.

They were right! Most of their bugs would show up randomly in a function after the 20th time that function was executed and still they weren't able to tell if the bug started before that function was called. Trying to figure out the root cause of the problem by running gdb in interactive mode was a pain. They would prefer looking at their source again and again to figure out the problem. Some of them would even rewrite their code using different (and usually more complex) program logic, hoping that the bug would go away.

I decided to have a look at the scripting capabilities of gdb and maybe try to help the students efficiently. I did find some useful information actually and managed to help a lot of students by showing them how to run gdb scripts. This whole experience motivated me to write this small tutorial on basic gdb scripting. Don't get me wrong, I secretly like the challenge of debugging in programming. Also, scripting gdb is really easy and it doesn't take that much time to teach. On the other hand, I would prefer spending my time as a tutor actually teaching people new material and not have my brain fried after trying to understand/debug some person's code for hours.

Tutorial

By default during startup, gdb executes the file `.gdbinit`. This is where you write your gdb code. In case you want to have many scripts that test different things, you can tell gdb to look at other scripts besides the default one by adding the `--command=<filename>` argument when running gdb.

So let's say that you want a backtrace every time a specific function is called. This is extremely useful when debugging recursive functions!. You would write the following code:

```
set pagination off
set logging file gdb.output
set logging on

# This line is a comment
break function_name
    command 1
    backtrace
    continue
end

run

set logging off
quit
```

So what happens here? I turn pagination off so I don't have to press Enter for every page of gdb output. I declare that I want to log all the output of gdb in a file called `gdb.output`. Then I start logging gdb's output and I set a breakpoint in my function. I use `command <breakpoint number>` (this case 1, because `function_name` is our first break point) to provide the commands that I want to be ran whenever gdb hits that breakpoint. As you can see I just print a backtrace and then continue until gdb hits another breakpoint. Now that everything is set, I run the program and when that finishes, I stop

logging the output and quit gdb. That's it! Now I can run gdb once and take a look at its output by opening `gdb.output`.

By the way, if your program takes certain arguments you can pass them as arguments in gdb. For example, run `gdb --args ./a.out arg1 arg2 ...etc`. Another way is finding the line that you execute `run` on your script and change it to `run arg1 arg2 ...etc`.

Most of the time though, you don't really need all this output. What if you needed to break in a function when a certain parameter of that function is passed a specific value? What if you needed to break to a certain line of code the 3rd time it is executed? You can do all these as I am going to show in the example below. So let's say, I want to break in function1 when one of its arguments, `param1`, is 32. I also want to break in line 142 of `file.c` when the variable `x` (which is in the same scope with the statement in line 142) is bigger than 4. Finally I want to set a breakpoint in `function2` and I want gdb to break the first 3 times this function is executed.

```
set pagination off
set logging file gdb.output
set logging on

set $var = 0 # yes, you can declare variables ...

break function1 if param1 == 32
    command 1
    print param2
    print param3->member1
    continue
end

break file.c:142 if x > 4
    command 2
    print y
    call checker_function
    continue
end

break function2 if $var++ < 3
    command 3
    print $var
    backtrace full
    continue
end

run

set logging off
quit
```

That's it! This is the end of this tutorial. In my humble opinion the above are more or less enough for programs of little to medium complexity. You can go ahead and learn more advanced features of gdb or research dynamic tracing that has been slowly coming to Linux (most UNIX platforms have that already). Even if you don't program actively on your own for the time being, knowing how to run and script a debugger helps a lot in the improvement of open source programs. For example, if your favourite program suddenly crashes and you can send a crash/bug report, attaching a simple backtrace and other gdb output that displays extra information will save a lot of time for the developers that will try to fix your problem.

Some Final Notes

Before I finish this post, I just want to give a piece of advice when using gdb. If your program crashes for whatever reason and you have a rough idea where that happens, just look at the relevant part of code for a while to see if it makes sense. We often type things by mistake when we are absent-minded that compile and give off-by-one and other types of errors. If that part of code is complex just go through it using gdb in interactive mode, don't start writing/editing scripts right away that target a bug you just found. You may actually spend more time writing a script to find a bug than actually stepping through your program in your head or using a debugger.

I would also like to encourage writing more checker functions in your source code that check conditions (or print out debug info on the program output during development) than huge gdb scripts with complicated logic. Checker function in your source can be called directly from your program AND from your gdb scripts too (see 2nd example script). Besides, checker function provide more flexible access to the internals of your program and they are written in the same language as the rest of your program.

EDIT: A big thanks to Nick Zivkovic for pointing out some mistakes in the initial version of this post.