A Very Brief Introduction to gdb

Introduction to Parallel Computing
Updated September 24, 2017

1 An Introduction

The Gnu debugger or gdb is a program that can be used to help find bugs in your program. To use it, you should be sure that your program is compiled with the -g option:

```
$ gcc -g -Wall -o hello hello.c
```

The -g option will tell gcc to create a symbol table so that gdb can translate machine addresses into information that’s easier for humans to use. For example, if the program crashes while executing the machine language statement stored at address 0xffff123d, if there’s a symbol table, gdb can determine that this is line 208 of your source program.

The easiest way to use gdb is to run your program under its control. Let’s look at an example. Recall the linked list program with the buggy Delete function. It’s on the class web site in the file linked_list1.c. After compiling it with the -g option,

```
$ gcc -g -Wall -o ll1 linked_list1.c
```

we’ll start up gdb with the command

```
$ gdb ll1
```

and gdb will respond with something like this

GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Copyright (C) 2014 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:

Find the GDB manual and other documentation resources online at:

For help, type "help".

Type "apropos word" to search for commands related to "word"...

Reading symbols from linked_list_bug...done.

(gdb)

The details will depend on the particular system you're using. What's important for us is the prompt (gdb). When we see this, we can start typing gdb commands.

To start execution of the program, we can just type

(gdb) run

and the program will start running:

Starting program: /home/peter/classes/pp_ug/code/linked_list/ll1

Now we can run the program using the same input that we use when we start it from the command line:

Please enter a command (i, p, m, d, f, q): i
Please enter a value: 5
Please enter a command (i, p, m, d, f, q): i
Please enter a value: 8
Please enter a command (i, p, m, d, f, q): i
Please enter a value: 1
Please enter a command (i, p, m, d, f, q): i
Please enter a value: 7
Please enter a command (i, p, m, d, f, q): p
list = 7 1 8 5
Please enter a command (i, p, m, d, f, q):
So let’s try deleting 8 from the list.

Please enter a command (i, p, m, d, f, q): d
Please enter a value: 8
Please enter a command (i, p, m, d, f, q): p
list = 7 1 5
Please enter a command (i, p, m, d, f, q):

Now let’s break things by trying to delete something that isn’t in the list

Please enter a command (i, p, m, d, f, q): d
Please enter a value: 9

Program received signal SIGSEGV, Segmentation fault.
0x0000000000040087f in Delete (head_p=0x603070, val=9) at linked_list1.c:126
126 pred_p->next_p = curr_p->next_p;

So gdb is telling us that the program crashed with a segmentation fault or segfault. Recollect that this means the program tried to access memory that is outside its assigned range. It’s also telling us that the segfault occurred in the Delete function, the arguments to the function were head_p = 0x603070 (remember that a pointer stores an address) and val = 9. Most important, it’s telling us that the crash occurred in Line 126, which is the assignment

126 pred_p->next_p = curr_p->next_p;

This strongly suggests that one or both of the pointers pred_p and curr_p is invalid. That is, one of the addresses stored in these pointers refers to a memory location that is inaccessible. We can try to check this by printing their values:

(gdb) print pred_p
$1 = (struct list_node_s *) 0x603010
(gdb) print curr_p
$2 = (struct list_node_s *) 0x0

The address that’s stored in pred_p, 0x603010, looks OK — it’s close to the value of head_p, which is probably OK, since we’ve been using it repeatedly to insert nodes and print the list.

On the other hand, the value stored in curr_p, 0x0, is the dreaded NULL pointer value, and trying to access memory referred to by a NULL pointer is guaranteed to cause a segmentation violation.
So we clearly need to rethink our algorithm for the `Delete` function. This means it’s time to stop coding and sit down with a pencil and piece of paper. So let’s quit `gdb`:

(gdb) quit
The program is running. Quit anyway (and kill it)? (y or n) y

## 2 gdb Commands

The commands we used in our example were

- **run.** This will start your program. If your program has command line arguments, you should just add them after the `run` command. For example, if you ordinarily start your program with

  $ ./my_prog in out left right

  you would start the program in `gdb` with

  (gdb) run in out left right

- **print.** This will print the contents of a variable. `gdb` is pretty good at figuring out the type of the variable, and printing the value in a reasonable format. For most variables you can just type

  print <variable name>

- **quit.** Quit `gdb`.

Here are a few additional `gdb` commands:

- **continue.** Continue execution. If your program seems to be running forever, you can stop it and get back to the `gdb` prompt by typing `Ctrl-C`. If you subsequently decide you want to resume execution where you left off, you can type `continue`.  

  4
• **where.** Print a trace showing where you currently are in the program, and the sequence of function calls from **main**.

• **step.** Execute the next line of code, stepping into functions. So the function code is effectively interpolated into the calling function.

• **next.** Execute the next line of code, but now a function call counts as a single line of code.

• **break <line number>.** Stop execution when the code in <line number> is the next statement to be executed. An alternative is **break <function>** which will stop before executing the first executable statement in <function>.

**gdb** has a fairly extensive online help system. Just type **help.** Alternatively, google **gdb** on the web. The **gdb** website is at [http://www.gnu.org/software/gdb/](http://www.gnu.org/software/gdb/), but there are also **many** websites with tutorials and additional information.

In the last few years Apple has switched from **gdb** to another debugger, **lldb**. All but one of the commands listed above seem to work in more or less the same way with **lldb**. The sole exception is **where.** In **lldb**, the command is **bt** which is an abbreviation of **backtrace**.