Using GNU's GDB Debugger

Debugging A Running Process

By Peter Jay Salzman

Previous: Stepping And Resuming Next: Debugging Ncurses Programs

Debugging A Running Process

So far, we've debugged executables, with and without core files. However, we can debug processes too. Think about that -- we can debug a process that has already been started outside the debugger. There are two ways of doing this: Using command line arguments and using the `attach` command.

Download and read beer-process.c and its . Compile it, and run it as a background job in one console (or xterm). It'll simply print out the number of bottles of beer on the wall:

```
$ ./beer-process &
[1] 17399
p@satan$ 100000 bottles of beer on the wall.
99999 bottles of beer on the wall.
99998 bottles of beer on the wall.
99997 bottles of beer on the wall.
```

With Command Line Arguments

With the beer process running one console, start GDB in another console with an argument list of the executable and the process ID. The process ID should've been printed when you started the background process:

```
$ gdb beer-process 17399
Attaching to program: code/running_process/beer-process, process 17399
0x410c64fb in nanosleep () from /lib/tls/libc.so.6
(gdb)
```

Chances are overwhelmingly good that the process is in `GoToSleep()`. Print out a backtrace and take a look at the stack:

```
(gdb) bt
#0  0x410c64fb in nanosleep () from /lib/tls/libc.so.6
#1  0x410c6358 in sleep () from /lib/tls/libc.so.6
#2  0x0804841f in GoToSleep () at beer-process.c:32
#3  0x080483e0 in main () at beer-process.c:14
```

Aside: Note that `GoToSleep()` calls the C library function `sleep()`, and `sleep()`, in turn, calls the system call `nanosleep()`. As you know, all library functions (glibc on Linux) do their job by calling system calls. I'm a

little surprised to see the library and system functions listed in the call stack since I'm not using a debugging version of glibc. Weird.

At this point, the backtrace should be very familiar to you. But there's an important distinction. We didn't run this program from within GDB. We ran it from the command line, and then had GDB attach to an already running process.

Look at the output of the beer process: you should notice that the process has stopped! Whenever GDB attaches to a running process, the process is paused so you can get a handle on what the call stack looks like. Let's do some interesting things.

In my output above, **`i=9997`**. Yours is probably different, but nevertheless, you should be able to follow along with me. Let's verify the value of **`i`** by selecting the stack frame for **`main()`** and looking at its value:

```
(gdb) frame 3
#3  0x080483eb in main () at beer-process.c:15
15                      GoToSleep();
(gdb) print i
$1 = 99997
```

No surprises here. As you'd expect, we can use **`next`** and **`step`** (which takes us out of **`nanosleep()`** and **`sleep()`** respectively, putting us into **`GoToSleep()`**):

```
(gdb) next
Single stepping until exit from function nanosleep,
which has no line number information.
0x410c6358 in sleep () from /lib/tls/libc.so.6
(gdb) step
Single stepping until exit from function sleep,
which has no line number information.
GoToSleep () at beer-process.c:34
34      }
(gdb) bt
#0  GoToSleep () at beer-process.c:34
#1  0x080483eb in main () at beer-process.c:15
```

Looking at the code, the next things to happen are that **`i`** will be decremented and then **`PrintMessage()`** will print **`99996 bottles of beer on the wall.`** However, suppose we wanted more beer? Let's change to the stack frame for **`main()`** (where **`i`** lives) and change the number of beers on the wall.

```
(gdb) frame 3
#3  0x080483eb in main () at beer-process.c:15
15                      GoToSleep();
(gdb) set var i = 99999999
```

Now quit GDB. When GDB detaches from the process, the process will continue along its merry way. We could also use the **`detach`** command to detach from the process without quiting GDB; I'll explain **`detach`** in the next session.

```
(gdb) quit
The program is running.  Quit anyway (and detach it)? (y or n) y
Detaching from program: code/running_process/beer-process,
process 17399
```

but with the new value for **`i`**:

```
$ ./beer-process &
[1] 17399
p@satan$ 100000 bottles of beer on the wall.
99999 bottles of beer on the wall.
99998 bottles of beer on the wall.
99997 bottles of beer on the wall.
99999998 bottles of beer on the wall.
```

```
99999997 bottles of beer on the wall.
99999996 bottles of beer on the wall.
99999995 bottles of beer on the wall.
99999994 bottles of beer on the wall.
```

I hope you're impressed by this! We attached GDB to a process that was already running. The process halted and we were able to do everything that we would've been able to do had we started the process from within GDB. Now that's power!

One non-debugging use I've had for this in the past is with scientific programming. I had PDE solvers and Monte Carlo applications that would run for a very long time. Whenever I wanted to take a look at how my simulation was doing or what some of the intermediary answers looked like, I'd attach to the process using GDB and inspect my variables. This was a much better option than simply printing everything of interest out, which could've possibly have taken hundreds of megs of disk space!

With The Attach Command

We can also debug an already running process using GDB's **`attach`** command to attach to a running process. Again, once attached, we can use the **`detach`** command to detach from the process.

If you quit the running background process from the previous section, restart beer-process in the background. Start GDB with no command line arguments. But use the **`attach`** command to attach to the running process.

```
$ gdb
(gdb) attach 17399
Attaching to process 17399
Reading symbols from code/running_process/beer-process...done.
0x410c64fb in nanosleep () from /lib/tls/libc.so.6
(gdb)
```

As before, the process should halt. This is when you do whatever it is you want to do with the process: debug, snoop, spy, modify, etc. When you're done futzing around, quit GDB:

```
The program is running.  Quit anyway (and detach it)? (y or n) y
Detaching from program: code/running_process/beer-process,
process 17399
```

As before, once you detach from the process, it'll continue running.

Processes Without Debugging Symbols

As with debugging executables and corefiles, it's only convenient to debug processes that were started from executables with debugging information compiled into them. To see this in action, strip the executable and run it in the background again:

```
$ strip beer-process
$ ./beer-process  &
[1] 32262
p@satan$ 100000 bottles of beer on the wall.
99999 bottles of beer on the wall.
99998 bottles of beer on the wall.
```

Debug the process and look at the call stack:

```
$ gdb
(gdb) attach 32262
Attaching to process 32262
Reading symbols from code/running_process/beer-process...(no debugging symbols found)...done.
(gdb) bt
#0  0x410c64fb in nanosleep () from /lib/tls/libc.so.6
#1  0x410c6358 in sleep () from /lib/tls/libc.so.6
#2  0x0804841f in ?? ()
#3  0x00000003 in ?? ()
#4  0x0001869d in ?? ()
#5  0xbffff7b8 in ?? ()
#6  0x080483eb in ?? ()
#7  0x0001869d in ?? ()
#8  0x0001869d in ?? ()
#9  0xbffff844 in ?? ()
#10 0x4102e7f8 in __libc_start_main () from /lib/tls/libc.so.6
#11 0x41150fcc in ?? () from /lib/tls/libc.so.6
```

Exercises

1. Suppose you're playing a game that you have source code for, like Doom, Nethack, or Duke Nukem 3D. How can you use GDB to cheat, like giving yourself extra health? If you wrote your own game, can you protect the integrity of networked games from people who would cheat like this? What kinds of things could you do?
2. Now suppose you're playing a game for which you do *not* have the source code. Can you still cheat in this manner? If so, how would you go about it?
3. Do a Google search on an application called "kcheat". Read the documentation. This person, in effect, wrote a debugger. If you have spare time, download the source and try to learn how it works. Browse the man page for the function `ptrace()`.
4. From the previous exercise, GDB could be considered as a "front end" to `ptrace()` system call. Look at `ps aux`. Do you see any processes that, if attached to with GDB, would be a security issue? Could cause a system to go down? Cause filesystem corruption? You probably have a process called "init" that has a process id of 1. Try to attach to it. Now become root and try to attach to it. There are some things that even root can't do!

⇦ Back: Stepping And Resuming

Up to the TOC

Email comments and corrections

Printable version

Next: Debugging Ncurses Programs ⇨